NIRS

Network Intrusion Detection & Response System

Group Report

MSc DCNDS Group 4
Department of Computer Science, University College London
Gower Street, London

Adedayo Adetoye
Andy Choi
Marina Md. Arshad
Olufemi Soretire

Supervisor: Steve Hailes

September 2003

TABLE OF CONTENTS

LIST OF TABLES

Table **Pages**

LIST OF FIGURES

Figures **Pages**

ABSTRACT

As computer networks become increasingly complex and network denial-of-service (DoS) attacks become more common place, automated network intrusion detection and response will be critical in providing reliable network services. In this report, we review the state-of-the-art in network intrusion detection systems and then propose a strategy of using network traffic profiles as the foundation for detecting and responding to network denial-of-service attack. We also present our implementation of the strategy and evaluate it under a controlled environment. The results show that our approach is effective in mitigating such attacks.

1 Introduction

Valuable properties need to be protected against theft, damage or destruction. People spend a lot of time, money and effort to ensure that their valuables and treasures are safe. Modern homes are installed with expensive alarm systems that can detect burglars, notify authorities when a break-in has occurred. Some alarms are even able to alert the owner when the house is on fire. Cars too are equipped with expensive alarm systems to safeguards against break-ins.

The same considerations should also be given to computer systems and data. Today's information systems in many organizations are highly interconnected via local area and wide area computer networks. These networks are potential targets of attack such as network bandwidth attack. One of the attacks, Denial of Service (DoS) has always been a critical threat to networks because of how easy it is to launch the attack from the many readily available attack tools. Malicious users look for vulnerable targets such as un-patched systems and networks running insecure services to launch the attacks. This method of attack has been known for some time but defending against it is a different matter.

Let's consider a scenario to paint a picture of the dangerous nature of this attack. An attacker sends forged ICMP echo packets to the broadcast address of a vulnerable network. All the systems on this network reply to the victim with ICMP echo replies. This rapidly exhausts the bandwidth available on the target network thus effectively denying service to legitimate users.

When a network is attacked, the ideal response would be to stop the attack before it can cause any further damage and deny users the services provided by the network Currently, the defence against DoS attacks relies heavily on intensive manual work by the network administrator. The first activity involves the use of network traffic probes and statistics. The second activity involves inserting packet filtering or rate limiting rules into the associated router. After which, the network administrators will

contact his counterpart in the upstream organization(s) from which the offending traffic is being forwarded to try and stop the attack. Obviously, this procedure has a major drawback in that it is time consuming and requires the administrator to be available immediately and always-on alert.

In this paper, we propose the framework and implement the design for a Network Intrusion Detection and Response System (NIRS) that automates this procedure. It employs a rate based learning agent approach to detect DoS attacks and applies a rate limiting *pushback mechanism* [1] to provide an active response in which the firewall, intrusion detection system (IDS), routers and other network components interact together to throttle the attack as close to the source as possible.

This report provides in the first part: the background, objectives, scope, tools and technologies used for the project. The second part describes the methodology and detailed implementation of the system and the testing environment. The third part contains the test result and the analysis of the result. The last part contains an evaluation of the project, discusses related work and highlights areas for future work

2 Background

2.1 Denial-of-Service Attack

A Denial-of-Service (DoS) attack is an incident in which a user or organization is deprived of the services of a resource they would normally expect to have. Typically, the loss of service is the inability of a particular network service to be available or the temporary loss of all network connectivity and services. Many attackers use other unsuspected third party computers, known as zombies or slaves, to flood the victim with millions of incoming traffic. A DoS attack is a type of security breach to a computer system that does not usually result in the theft of information. However, these attacks can cost the target organization a great deal of time and money.

2.2 Intrusion Detection

Intrusion detection has been an active field of research for about two decades. This is exemplified by an influential paper, published in 1980, *Computer Security Threat Monitoring and Surveillance* by James Anderson [2]. It was followed some years later in 1987 by the seminal paper *An Intrusion Detection Model* by Dorothy Denning [3] that provides a methodological framework for an intrusion detection system.

An intrusion detection system (IDS) inspects all inbound and outbound network activity and identifies suspicious patterns that may indicate a network or system attack from someone attempting to break into or compromise a system. Network Intrusion Detection Systems (NIDS) monitors packets on the network wire and attempts to discover if an attacker is attempting to break into a system or cause a denial-of-service attack.

2.2.1 IDS Components

In [4] the components that make up an Intrusion Detection system were identified as follows:

- Information Source – Data utilised by the IDS
- Analysis Engine – Process by which the intrusion decision is made
- Response – action taken when an intrusion is detected [4].

2.2.2 Detection Techniques

IDS falls into three different categories:

Host-based vs. network-based systems

In a host-based system, the IDS examines the activity of each individual host in the system while, network-based intrusion detection systems are dedicated software systems that sit on a network wire and analyse the individual packets flowing through a network. They can detect malicious packets that are designed to be overlooked by a firewall's filtering rules.

Anomaly detection vs. misuse detection

Denning [3] described the classical model for anomaly detection:

- A model is built which contains metrics that are derived from system operation.
- A metric is defined as a random variable representing a quantitative measure accumulated over a period. Example: average CPU load, number of network connections per minute.
- Security violations could be detected from abnormal patterns of system usage.

In anomaly detection, the network administrator defines the normal behaviour of the network's traffic, protocol and typical packet size. The anomaly detector works by monitoring network segments to look for anomalies by comparing their state to the normal behaviour that has been defined.

In misuse detection, the IDS analyzes the information it gathers and compares it to large databases of attack signatures. It looks for signatures of specific attack type that has already been documented. This system relies heavily on a good attack signatures database as the base to compare the packets against.

Reactive system vs. passive system

The IDS in a reactive system responds to suspicious activity by logging off a user or by reprogramming the firewall to block network traffic from the suspected malicious source while the IDS in a passive system detects a potential security breach by logging the information and signalling an alert.

NIRS is a hybrid of three techniques – anomaly, reactive and network based system which detects intrusions by observing deviations from normal behaviour and re-programmes a firewall to block the malicious traffic.

2.2.3 False Positive and False Negative

The main goal of an effective IDS as observed in [5] is to provide high rates of attack detection with very small rates of false alarms. There are two types of false alarms associated with Intrusion detection systems i.e. False positive and False negative.

As highlighted in [6], false positives occurs when the IDS sensor misinterprets normal packets or activities as an attack. Such errors can degrade the productivity of the systems because they can invoke unnecessary countermeasures. On the other hand,

false negative errors occur when an attacker is misclassified as a normal user. A fatal problem may arise from a false negative error as unauthorized or abnormal activities generate unexpected or undesirable operations of the systems. This can cause great losses for an organization.

2.3 Active Response

For accurate intrusion detection, the system must have reliable and complete data about the target system's activities and this is a complex issue in itself. Most systems have logs generated by either the operating system routers or firewalls that provide information on network operation and activities. However, logging too much information can put a serious overhead on the system. Therefore, the amount of system activity information collected is a trade-off between overhead and effectiveness.

The concept of active response in an IDS is based on the idea of having an IDS capable of automatic reaction once an attack is detected. The goal is to prevent the attack from spreading to other parts of the network and using up all of the network bandwidth and hindering other network services.

A paper in SecurityFocus group [7] identified two types of response mechanism:

1. Session Sniping
2. Firewall Update

2.3.1 Session Sniping

Session sniping or knockdown is a direct intervention between an attacker and the victim in order to disrupt the communication. To do this, the IDS sends packets to break down the connection that triggered the response. The most effective way to

knockdown a TCP connection is to forge packets to reset the connection. This is done by sending forged packets with the TCP Reset bit set to one on both systems.

2.3.2 Firewall Update

A second method mentioned in [7] is firewall rules manipulation. An IDS uses this method to actively respond to attackers by instructing the firewall to drop or block all traffic coming from the source IP of the attacker. The reasoning behind this method of response is to stop the attacker from further doing damage to the network.

2.3.3 Pushback mechanism

Another mechanism used for response in intrusion detection system is pushback [1]. Pushback mechanism is based on aggregate congestion control (ACC) and its particularly used in defending against Distributed DoS. It allows a router to request adjacent upstream routers to rate-limit traffic corresponding to the specified aggregates. Pushback can prevent upstream bandwidth from being wasted on packets that are eventually going to be dropped downstream. In the case of DoS attack, if the attack traffic is concentrated at a few upstream links, pushback can protect other traffic within the aggregate from the attack traffic.

2.4 State-of-the-Art

This section briefly describes some of the recent and current intrusion detection research effort. A lot of the current research has been focusing on anomaly based detection. There are many different approaches to anomaly based intrusion detections that can be found.

Some of them are:

- Probability based
- State based
- Data Mining

2.4.1 Probability Based Approaches

Anomaly detection systems such as SPADE [8], ADAM [9], and NIDES [10] adopted an approach in which the system will learn a statistical model of normal traffic, and flag deviations from this model. These statistical models were usually based on the distribution of elements such as source and destination addresses and ports per transaction (TCP connections, and sometimes UDP and ICMP packets). The lower the probabilities, the higher the anomaly scores were, since these are presumably more likely to be hostile.

ADAM used a classifier which could be trained on both known attacks and on (presumably) attack-free traffic. Patterns which did not match any learned category were flagged as anomalous. ADAM also modeled address subnets (prefixes) in addition to ports and individual addresses. NIDES, like SPADE and ADAM, modeled ports and addresses, flagging differences between short and long term behaviour.

SPADE, ADAM, and NIDES all used frequency-based models, in which the probability of an event was estimated by its average frequency during training. While PHAD [11], ALAD [12], and LERAD [13] used time-based models, in which the probability of an event depended instead on the time since it last occurred. For each attribute, they collected a set of allowed values (anything observed at least once in training), and flagged novel values as anomalous.

2.4.2 State Based Approach

Another slightly different approach to pure anomaly is the state based approach to network intrusion detection. A recent paper by Sekar et al. titled *Specification-based Anomaly Detection: A New Approach for Detecting Network Intrusions* [14] described the approach in which it tried to detect intrusion through anomalous state transition and at the same time incorporate state machines of network protocols. The main advantage of this approach is that it can detect high rate of known and also unknown attacks. At the same time, it has a rather reasonable false alarm rate that is comparable to the misuse methods. However it comes at a heavy price of having to build complex state based models of network protocols in the network.

2.4.3 Data Mining Approach

In their paper *Mining in a Data-flow Environment: Experience in Network Intrusion Detection* [15], Lee, Stolfo and Mok described the Data Mining approach. Data Mining looks at connection sessions and this is different from the normal anomaly approach which looks at individual packets. This approach works by using data mining tools and methods to differentiate anomalous sessions from normal sessions in an iterative manner using training data it gathers as a reference.

3 Objectives and Scope

3.1 Motivations

In the past few years, network based computer systems have been playing an increasingly vital role in modern society [16] and we have witnessed a tremendous growth in the inter-networking arena with the Internet and the Web infiltrating all segments of the economy faster than any previous technology. Though the possibilities and opportunities seem limitless; unfortunately however, the risks and incidences of security breaches are also on the increase. In [17], it was noted that during the past twelve years, the growth of incidents reported to the Computer Emergency Response Team/Coordination Center (CERT/CC) has reflected the growth of the Internet itself. Figure 3.1 below from [17] which shows the number of incidents reported to CERT/CC between 1988 and 1999 illustrates this growth..

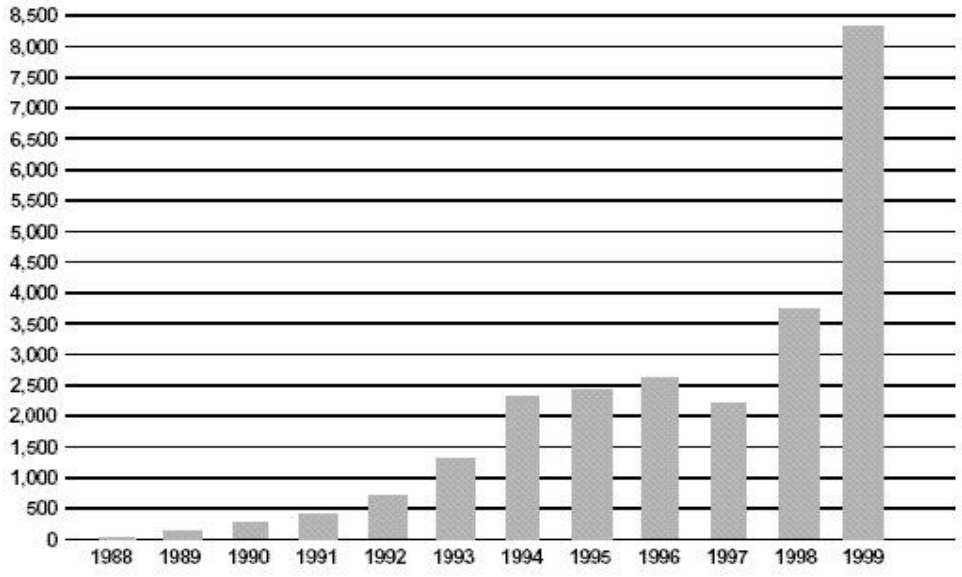

Figure 3.1: Growth In Number Of Incidents Handled By The CERT/CC

As indicated in [18], while a computer system should provide *protection*, *integrity* and *assurance* against denial of service, however, due to increased connectivity (especially on the Internet), and the vast spectrum of financial possibilities that are opening up, more and more systems and networks are subject to attack by intruders.

Highlights of the year 2002 annual Computer Crime and Security Survey" [17] conducted by the Computer Security Institute (CSI) with the participation of the San Francisco Federal Bureau of Investigation's (FBI) Computer Intrusion Squad indicated that; ninety percent of respondents (primarily large corporations and government agencies) detected computer security breaches within the last twelve calendar months, Eighty percent acknowledged financial losses due to computer breaches and forty percent detected denial of service attacks. This confirms that the threat from computer crime and other information security breaches continues unabated and that the financial toll is mounting.

While it is very important that the security mechanisms of a system are designed so as to *prevent* unauthorized access to system resources and data, completely preventing breaches of security appear, at present, unrealistic [18]. Thus, a response to these growing threats is for organizations to put in place a layered security architecture to achieve optimum [17]. As noted by Kumar Das [18], Intrusion detection has become an essential component of computer security in recent years with Security Administrators complementing existing security measures with intrusion detection systems (IDSs) to achieve defence in-depth [17].

The Intrusion Detection research field effort is focused on detecting intrusion attempts so that action may be taken to throttle them. There are currently, a good number of commercial and research intrusion detection systems that detect intrusions using either the misuse or anomaly detection paradigm. However, most of these systems employ a passive approach to responding to detected intrusions. This normally involves sending an alert to an administrator to notify him of the detected intrusion or logging to a file. It is obvious that this approach of relying on a Systems Administrator to respond to detected intrusion does not scale, and in most cases could

not tackle problems as soon as they occur or are about to. A more effective approach is required. Automatic active response to intrusion has thus recently become an active research area.

Therefore, the motivation for this work is two fold:

1. to contribute to the evolving field of implementing an active response to intrusion detection
2. to explore the use of traffic profiling approach to intrusion detection

3.2 Project Goal

The goal of this research effort is to design and implement an Intrusion detection and response system which provides end-to-end network security from intrusion detection to active response.

3.3 Project Scope

The project will define a framework for detecting and responding to network based Denial of Service type attack. It will also implement a network traffic profile-based Intrusion Detection System for detecting DoS attacks. Finally, it will implement an active response system using the pushback mechanism for rate limiting. Due to time constraints the security requirements of the system like authentication of agents, or inter-agent communication protocols and system policy description or specification will not be considered.

4 Methodology

Denial of Service (DoS) Intrusion detection mechanisms fall into two broad categories in their mode of implementation, i.e. Misuse detection and Anomaly detection. In misuse detection, the IDS looks for recognised attack patterns (signatures) that has already been documented in its database. Anomaly detection systems on the other hand look for deviations in normal usage behaviour patterns. It basically monitors network segments to compare their state to the normal baseline and look for deviations. While misuse patterns are often simpler to process and locate, they tends to fail when new attack methods are discovered and implemented. Anomaly detection on the other hand though able to detect new attacks, is often highly difficult to implement, as what is "normal" usage has to be established and it must be tailored to the environment it is being deployed in as behaviour patterns and system usage often vary widely in different environments.

The approach adopted in this work falls in the Anomaly detection domain. We employed a based Rate based premise to characterise normal network traffic and detect intrusion from deviations from the characterised traffic profile.

Furthermore, we also investigated the possibility of using a Token Bucket implementation to detect intrusions given a traffic characterisation derived from a linear bound arrival process algorithm. A detailed description of both follows.

4.1 Rate-Based Traffic Profiling

The traffic profile of the network (with the network isolated from external traffic) was first generated via a learning process. This information was then used to set the threshold of the detection engine. The detection engine then monitors and compares network traffic (when the network is not isolated) with the set threshold values and

sends an alert when the values are exceeded. The alert serves as a trigger to activate the response system which then employs the pushback paradigm [1] to throttle the attack by rate limiting along the identified attack path. A more detailed description of the different component follows later.

4.1.1 Network Profiling

The first step in our approach was the learning process in which a learning agent gathered statistics of the usage of the network in an isolated network environment. The data gathered were the average data rate per protocol, average packet rate per protocol, average packet size per protocol and the standard deviation of these measurements from their respective means. While theses data can be gathered for different protocols, we gathered them for the three main protocols used in DoS attack i.e. TCP, UDP and ICMP protocols. These parameters which indicated the contribution per protocol were used to build the network profile and were inputs for the detection and the response agent.

4.1.2 Intrusion Detection Model

Given that the learning agent has gathered a normal usage profile for the network, the next step was to use that information in helping to detect when the network is under a DoS attack. Using Figure 4.1 we want to describe the steps that form the foundation for the detection framework developed in the NIRS project.

Assuming for a protocol, the average data rate is given by **μ**, which was measured over the network profiling period, and $\mathbf{y_{max}}$ is the maximum data rate available for that protocol constrained by the actual network data rate, and σ is the standard deviation of the population used for the profile. We want to find a function **P(x)** that

gives the probability that an observed measurement during the network runtime constitutes a DoS attack traffic.

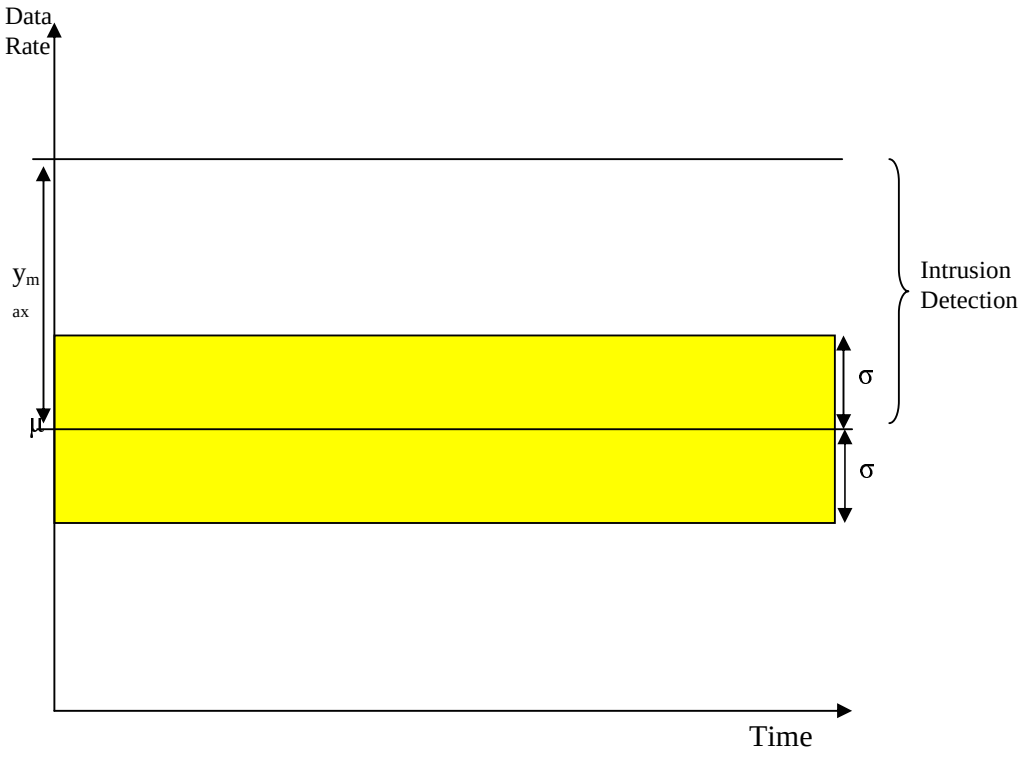

Figure 4.1: Network Profile Parameters Used in Intrusion Detection

By definition, DoS attacks render a service unusable by consuming all the resources available to the service. In this context we are interested in network DoS, that is, a complete consumption of network bandwidth that renders the network unusable to other legitimate traffic. We want to associate a probability **P(x)** =1 with a condition when all the available bandwidth is consumed, i.e. data rate at that instance is equal to $\mathbf{y_{max}}$. We want to start looking for suspicious traffic profiles when values exceed the mean **μ**, as such we associate a probability **P(x = μ)** =0 where the measured value equals the population mean for the profile sample.

The level of confidence on the mean is a function of the population size used during the network profiling period; the larger the size the more accurate the mean tends to be, furthermore, the nature of the traffic over the sample period also affects the interpretation of the level of confidence we have over samples whose values are close to the mean. For traffic profiles that are bursty over the sample period, there would tend to be larger variations about the mean, which is reflected by the variance **σ** of the population about the mean. Steady rate traffic would tend to have lower variances in comparison with bursty traffic. As a result the variance of the population affects the value of **P(x)** close to **μ**. For two three-tuple profiles $\{P_1, \mu, \sigma_1\}$ and $\{P_2, \mu, \sigma_2\}$, generally

$$P_{1:x\to\mu^+}(x) > P_{2:x\to\mu^+}(x)$$

Where

$$\sigma_1 < \sigma_2$$

If we represent the probability function with a sigmoid, we have:

$$P(x) = \frac{1}{1+e^{-ax}}$$

This is shown in Figure 4.2

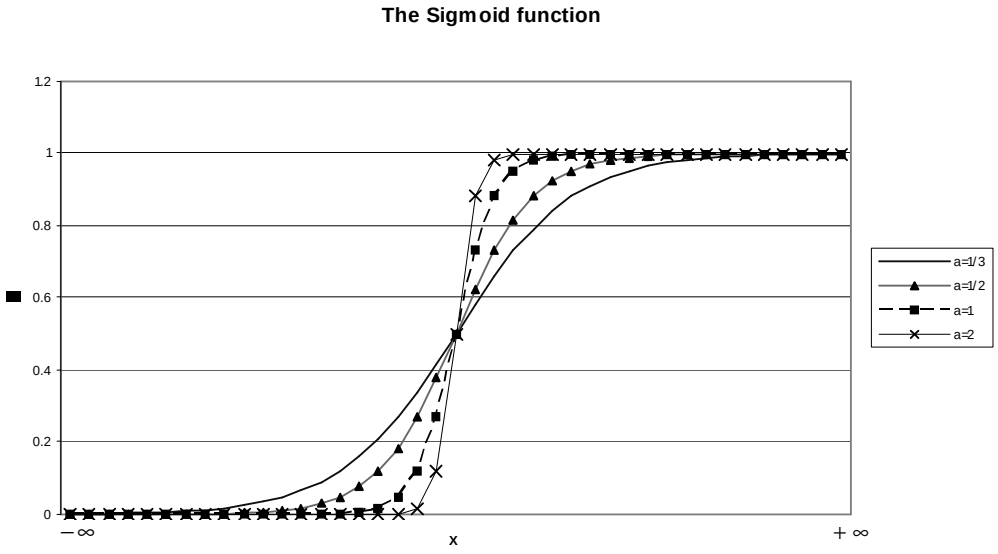

Figure 4.2: The Sigmoid Function Showing the Effect of *a* on the Slope

As shown in the figure, curves with a larger values of ***a*** have steeper slopes than those that do not. It can be shown that ***a*** is proportional to **σ.** By normalizing **σ** over $\mathbf{y_{max}}$ we set $a = \sigma/y_{max}$ which gives:

$$P(x) = \frac{1}{1 + e^{-f\left(\frac{\sigma}{y_{\max}}\right)x}}$$

The probability **P(x)** indicates the level of confidence that a network traffic sample at an instance constitutes an attack or not. However, to account for possible transients spikes that may lead to false positives; a parameter is introduced to relax P(x) measurements over a number of steps. This parameter leads to the confidence threshold beyond which a signal is sent to the response system of the probability of intrusion. The confidence measurement is thus given by

$$c_n = \frac{1}{n}\sum_{i=1}^{n} P(x_i)$$

where n is the number of steps before a trigger signal is sent to the response system. Given a confidence threshold of **C**, a trigger signal is sent if and only if $x_i > \mu$ for every $i \in \{1, 2, \ldots, n\}$ and $c_n \geq C$. The trigger signal **S** consists of a level of confidence and network information parameters. That is

Iff

$$x_i > \mu \quad \forall i \in \{1,2,...,n\}$$

and

$$c_n = \frac{1}{n}\sum_{i=1}^{n} P(x_i) \geq C$$

then

$$S=\{c_n, \text{NET\_INFO}\}$$

Where NET_INFO contains network specific information like the offending protocol, IP addresses, interface etc required by the response system.

4.2 Token Bucket Approach

Token Bucket descriptor belongs the class of Linear bounded arrival process, or LBAP, which is a class of data source descriptor. As pointed out in [19], the LBAP descriptor correctly model the fact that even a "smooth" source may have periods in which it is bursty [19]. Even though it may not accurately represent sources that have occasionally very large burst, it does gives a better characterisation of the source then the aggregated rate in most cases.

Hence, we investigate the use of token bucket parameters, instead of aggregated rate, to profile *normal* traffic destined to a particular host. Given this profile, we can then use a token bucket policer to identify if the incoming traffic, at any point in time, has

deviated from the *normal*. A decision of DoS attack can then be made by how much deviation from *normal* is accepted as noise.

4.2.1 Token Bucket Policer

Linear bounded arrival process, or LBAP, is a class of data source descriptor. A LBAP descriptor for an arbitrary data source describes the number of bit the source transmits in any given interval of length *t* by a linear function of *t*. This linear function can be characterised by two parameters σ and ρ, so that:

Number of bits transmitted in any given time interval $t \leq \rho t + \sigma$

ρ corresponds to the observed long-term average rate of data generated in the network by the source, and σ is the longest burst a source would send given a choice of ρ while still obeying the above definition.

A token bucket/leaky bucket regulator regulates a data source to a LBAP descriptor [20] Formally, it accumulates fixed-size tokens in a token bucket and transmits a packet only if the sum of the token sizes in the bucket adds up to the packet's size. On departure of each packet, the regulator removes tokens corresponding to the packet size from the token bucket. Token are refilled periodically to the bucket at rate ρ and the size of the bucket is limited by σ. The regulator would delay a packet by queuing it in a data buffer if it does not have sufficient token for transmission.

If a token bucket regulator does not have a data, it is called a token bucket policer [19]. Consider if we attach such a policer to a data source, we can then verify the conformance of the traffic generated by that source to a particular token bucket descriptor, which is a form of LBAP descriptor.

Token bucket regulators and policers are widely used in both academic and industrial settings. For example, token bucket forms the foundation of Quality-of-service(QoS)

guarantee for both Asynchronous Transfer Mode (ATM) networks and INTSERV and the QoS framework for the internet put forward by the IETF. In both cases token bucket policers are adopted to "police" if data generated by a particular source does not conform to the LBAP descriptor negotiated during the "call setup" process.

4.2.2 Traffic Characterization and Policing

For a given data source, we would like to come up with a LBAP descriptor for that source such that there exists no other descriptor which has a smaller ρ and σ. We call such a descriptor the minimal descriptor, but it is not unique for any data source. In fact, the set of minimal descriptor can be described as the *burstiness curve* for the data source [21]. Each combination would require a different data buffer size for token bucket regulator. However, it does not matter for traffic policer because a policer does not have any data buffer by definition. Thus, for the purpose of traffic policing, it is indifferent to use any point on the *burstiness curve*.

As a result, in order to characterise a data source for our purpose, we will only need to find a point on the corresponding *burstiness curve*. In fact, two algorithms have been suggested in [22] to compute the burstiness curve of video sources. It has been further claimed that such algorithm is suitable for any bursty ON-OFF traffic, including voice and data. With the *normal* traffic characterised by picking a point, (ρ', σ'), on the curve generated by the algorithms, we can then use the pair as a basis to configure the token bucket traffic policer.

4.3 The Response System

The agent-based response system uses the information supplied by the intrusion detection framework to carry out a response using the pushback mechanism. The pushback mechanism is a method for isolating offensive traffic by iteratively reducing

traffic destined for a resource, walking from the destination resource back towards the traffic source; hence the name pushback [1]. Given the two-tuple, $S=\{c_n$, NET_INFO}, the response agent may use this information as a basis for further forensic examination at the current site to further refine the quality of isolation of the offending traffic from normal traffic. Within the constraints of the available information, the response agent sends a pushback message to upstream agents along the path of the attack traffic to rate-limit on the attack traffic given the description gathered from the detection engine and, if available, results gathered from further forensic analysis extracted after the intrusion was detected.

To limit the performance overhead associated with intrusion detection, the vital statistics that the detection system looks at are minimized, however in the event of the statistics being gathered indicating a high probability of intrusion, the response system may ask for further forensic analysis which involves looking at non-spoofable attributes of the traffic like the network interface from which the traffic is originating, destination IP address, protocol header options etc.

In a rate-limiting scenario, the NET_INFO specification could involve the protocol, protocol header options, destination IP address, network interface, data rate and source IP. The problem with specifying source IP as a classification parameter arises when the system is responding to streams from unauthenticated sources. In such instances, the source IP might be spoofed and the rate limit would be applied to innocent sources, this might translate to a very important customer IP address. There is also the issue of intrusion detection and automatic response system themselves being used as an attack tool against the system they are protecting. For example, an automatic response system that blocks all offending source IP addresses permanently or semi-permanently may be used by runaway hackers to cripple legitimate services by spoofing customer IP addresses during attack. This issue is beyond the scope of this project. Also but source IP addresses were not used as a classification parameter during response.

4.4 Traffic Generator

4.4.1 Choice of user traffic

We have decided to validate our DoS attack detection and response approaches under two types of user data source. The first type is a bursty one. In particular, we considered a data source which transmits a sequence of packets for a particular period, also known as “busy” period, and then becomes “silent”, a period which the source generate no network activity, for a relatively longer period. Typical example of a data source which exhibits such a traffic pattern is HTTP request generated by a user. Given web traffic remains the dominant traffic in the internet today, it is very important to include such profile into our study. The second type of data source is one which generates fix-sized packets at a fairly constant rate. Example of this type of data source includes, multi-media streaming and conferencing session (without any host-based conditioning techniques such as compression). Since there is a high interest in deploying voice and multi-media traffic over internet, we felt that it is useful to include this type of data source in our studies.

4.4.2 Simulated User Network Traffic

We have attempted to generate user network traffic using VideoLAN to stream a small video clip over our experimental overlay network. Indeed we have tried this setup for several times and each trail was haunted by three major technical problems.

First, our network was build by connecting UML instances hosted on four separate physical Linux machines. Unfortunately, these Linux machines were powered by only a single Pentium II – 233 MHz CPU. Running four instances of UML while forwarding packets at the rate VideoLan generated proved to be much for the machines to handle. Large number of packets were dropped by the intermediate routers. We have tried to use video clips with fairly low resolution and size, but this

problem persisted. We have considered using audio streaming applications, which use much lower bandwidth and generate packets at a relatively lower rate. However, such applications required access to the physical device (usually /dev/dsp in Linux) and virtualization of such device is not possible under the current UML implementation.

Secondly, even if we managed to find a streaming application which generates packets at a rate acceptable to the experimental setup, we still face the problem of playing out the streamed packets. As mentioned before, audio has already been ruled out. In order to play out any video from inside an UML environment, we need to establish a connection between the X server on the physical host, which has exclusive access to the frame buffer device, and the X client application which tries to display the received packets. The only possible connection for such a purpose is to connect dummy X server (XNest, as described in Chapter 5) inside UML to the physical host via UDP. The side-effect of such a setup is that the X protocol would generate a significant amount of traffic at the ethernet interface of the UML hosts and this would affect our experiment which is based on measuring UDP traffic.

Thirdly, while VideoLAN is extremely useful as a demonstration tools to show the effectiveness of the detection and response systems through the change in visual quality, it does not produce any measurable numeric values. Without such numeric values, it is very difficult to analyse qualitatively the Receiver Operating Characteristics (ROC) of the detection system as well as the effectiveness of the response system.

We realised that these two requirements can be easily satisfied by using a packet generator and a corresponding receiver. Using a packet generator, we were able to specify the size and the inter-transmission time of each packet. Hence, we can engineer the source such that the limit imposed by the physical machines is not exceeded. Furthermore, between the packet generator and receiver, it is possible to work out how many packets have been dropped in a particular session. This number would be helpful in evaluating the response system.

4.4.3 Simulated Attack Traffic

We have studied three ways to perform an actual DoS attack, either via simulation or otherwise. First, we have tested two commonly used attack tools, tfn2k and stacheldraht. While we failed to compile the version of stacheldraht we obtained, we were able to compile tfn2k after making some modifications to the source. With that, we had successfully launched a DoS attack against one of our test machines using tfn2k. The effect was indeed devastating: the network segment became totally unusable.

However, tfn2k requires direct interaction with the device driver, and we were not sure of how much interaction would affect our test environment. Since Linux does not have process QoS by default, we were concerned that tfn2k may take up too much processing power of the physical and thus degrade the "performance" of the test network.

As an alternative, we explored the possibility of using the simple UNIX programme, ping, to simulate attack traffic. With the **–f** flag, ping would output packets as fast as they came back, via ICMP ECHO_REPLY, or one hundred times per second, whichever is more [23].

Consider the case of one hundred ICMP ECHO_REQUEST packets per second. Each packet was less then 80 bytes and thus the minimum traffic generated would be 8000 bytes per second. This rate was too low to be useful to act as an attack source. For any rate above this, the actual ICMP packet generation rate would be directly related to how soon it received a reply. While this was good behaviour for normal usage, so that ping, even with **–f** supplied, would scale its rate according to the network condition, it was inconsistent with how a "good" bandwidth-abuse DoS attack source should behave. The objective of such a source is to flood the network by sending packets as fast as possible. Hence, we decided against using ping.

Finally, we also had the option to use our own packet generator to simulate an attack source. Even though it did not generate packet as fast as attack tools like tfn2k, but the rate it could generate already surpassed the rate that the experimental network can handle. Since we attempted to look at generic DoS attack which introduce abnormal level of network activity, we were indifferent on how the packets are generated.

At the end, we decided to use our own packet generator because it gave us more control in defining what packet generation rate constituted a DoS attack.

5 Tools and Technologies

This chapter describes the major tools and technologies used in the project.

5.1 User Mode Linux

User Mode Linux (UML) was originally developed by Jeff Dike on i386 architecture as an implementation of a user space virtual machine. UML consists of a Linux kernel that runs on a host Linux system, in a set of Linux processes.

UML is a port of the Linux kernel to itself. That is, it considers the Linux system call interface to be a platform just as Intel x86 architecture and it is a port of Linux to that platform. UML is just a Linux kernel that has been tweaked so that instead of talking to the bare metal, it talks to the services provided by a lower-level kernel. UML kernel runs as a process under a parent Linux session: it uses a separate partition (a loopback-mounted filesystem, stored in a file) as its root file system, and it doesn't share any processes, memory or files with the parent Linux session.

UML directly runs the host's unmodified user space. If processes run exactly the same way in a virtual machine as in the host, then their system calls need to be intercepted and executed in the virtual kernel. Each process within a virtual machine gets its own process in the host kernel. The host process is used solely as an execution context. It will have completely different attributes from the UML process, including different name, different uid, and different pid. Even threads sharing an address space in the user-mode kernel, get different address spaces in the host.

UML can run essentially anything that will run on the host (the exceptions mainly being things that deal directly with hardware). This means that a UML can do anything that a physical Linux machine can, with the advantage that UMLs can be

created and destroyed as needed. The fact that a UML instance is virtual gives it capabilities that make even more applications possible.

This project decided on using UML instead of a physical Linux machine because of limited number of machines available for the project and the flexibility it provides to configure a network tested with networking device functionalities e.g. a router without using the physical device. UML provided a solution as many instances of UML host could run on a physical machine.

5.2 OpenVPN

OpenVPN is a robust and highly configurable IP tunnelling program. It takes the approach of being a user-space daemon. Its primary use is for linking networks together by constructing multiple tunnels to or from the same peer.

UML supports no less than six different methods to provide networking to the virtual Linux system. The most used network transport is through the TUN/TAP interface on the host, which is used by OpenVPN. With a tap device, a virtual ethernet device can be created within UML, and all the traffic sent to UML instances will appear on the tap device on the host.

5.3 Xnest

UML does not have an X Windows and only runs on command line. To transform the virtual machine into a full blown Linux box, it need to run an X server. X windows is needed to display video streams from the VideoLan application. Xnest is the UML X server, it does not use a video card for its display but instead uses another X server.

5.4 Snort

Snort, an open source network intrusion detection system, it is a packet sniffer that monitors network traffic in real time by scrutinizing each packet closely to detect a dangerous payload or suspicious anomalies. It is based on *libpcap* (for library packet capture), a tool that is widely used in TCP/IP traffic sniffers and analyzers. Snort detects attack methods such as DoS or buffer overflow through protocol analysis and payload searching/ matching. When it detects a suspicious packet, it sends a real-time alert to either syslog, a separate alert file, or to a pop-up window. In this project, Snort is used in the token bucket based detection mechanism.

5.4.1 Snort Rules

Snort uses a simple, lightweight rules description language that is very flexible. The rules are divided into two logical sections, the rule header and the rule options. The rule header contains the rule's action, protocol, source and destination IP addresses and netmasks, and the source and destination ports information. The rule option section contains alert messages and an arbitrary-sized list of information on which parts of the packet should be inspected to determine if the rule action should be taken. This list of information was made up of either built-in snort directives or plugins. One simple rule is illustrated here:

```
alert icmp any any -> 192.168.1.17/32 any (msg:"ECHO
   request"; itype=8; dsize: "400<>500";)
```

The text up to the first parenthesis is the rule header and the section enclosed in parenthesis is the rule options. In this case, it described "an alert would be raise if there is any ICMP packet destining to a host with IP address 192.168.1.17 which matches **every** criteria described in the option section". The options, in this case, were that the ICMP packet header indicates the packet is type 8 (i.e. ICMP echo request); and the payload of the datagram is between 400 and 500 bytes. While each rule

contains elements which had a logical **and** relationship, snort can perform inspection on any number of such rules for each packet.

5.5 tc and Linux DIFFSERV

A tool used in the token bucket based response mechanism is tc. As part of Linux DIFFSERV [24] implementation, it is a utility used to interacte with the Linux kernel to configure different network quality-of-service (QoS) settings in the running kernel. It provided features such as traffic policing, shaping and DIFFSERV marking.

5.6 Iptables

UML comes with advanced tools for packet filtering which is the process of controlling network packets as they enter, move through, and exit the network stack within the kernel. The firewall program, called iptables, can restrict access by IP address, port number, interfaces or by the properties of the packets.

5.6.1 Packet Filtering

Traffic moves through a network in packets. A network packet, which is a collection of data in a specific size and format, contains information that helps it navigate the network and move towards the destination. The information includes the source address, the destination machine, the interface it should be going through or the packet type.

The UML kernel has the built-in ability to filter packets. Its IPTables consists of a series of tables. A default IPTables setup comes with three tables as shown in Table 5.1.

Table 5.1: Default Table for IPTables

Table	Usage
Filter	The default table that filters out packets, preventing them from coming in or going out. This is the most used table.
Nat	This table is used to alter packets that create a new connection
Mangle	This table has the capability of actually modifying packets according to various criteria.

Each of these tables has a group of built-in chains that correspond to the actions performed on the packet by IPTables. We will focus on the filter table because it is the table used to write rules for firewall reprogramming in the IDS and also counting packets for the learning engine. The built-in chains for the filter table are shown in Table 5.1 below.

Table 5.2: Built-in Chains For Filter Table

Chain	Usage
INPUT	This chain applies to packets received via a network interface.
OUTPUT	This chain applies to packets send out via the same network interface that received the packets.
FORWARD	This chain applies to packets received on one network interface and sent out on another.

5.6.2 IPTables Rules

Each chain consists of a sequence of rules. Each rule consists of a condition that may or may not be met, and a target to which the packet is sent if the condition is matched. If the condition is not matched, the packet is passed to the next rule in the chain. Regardless of the destination, when packets match a particular rule on one of the tables, they are designated for a particular target or action to be applied to them. Every chain has a default policy to ACCEPT, DROP, REJECT, or QUEUE the packet to be passed to user-space.

The common iptables commands have the following structure:

Iptables [-t <*table-name*>] <*command*> <*chain-name*> <*parameter-1*> <*option-1*> <*parameter-n*> <*option-n*>

The <command> option is the centre of the command. It dictates a specific action to perform, such as appending or deleting a rule from a particular chain, which is specified by the <chain-name> option. <parameter> define the way the rule will work and <option> will tell what will happen when a packet matches the rule.

5.7 Videolan

VideoLAN is designed to stream MPEG videos on high bandwidth networks. It has two parts:

- VLS (VideoLAN Server), which can stream MPEG-1, MPEG-2 and MPEG-4 files and live videos on the network in unicast or multicast.

- VLC (VideoLAN Client), which can be used as a server to stream MPEG-1, MPEG-2 and MPEG-4 files and live videos on the network in unicast or

multicast; or used as a client to receive, decode and display MPEG streams under multiple operating systems.

In the project demonstration, video will be streamed from the server to the intended client. Another machine will flood the network with packets to reduce the available bandwidth for the video stream.

An illustration of the complete VideoLAN solution is shown in Figure 5.1.

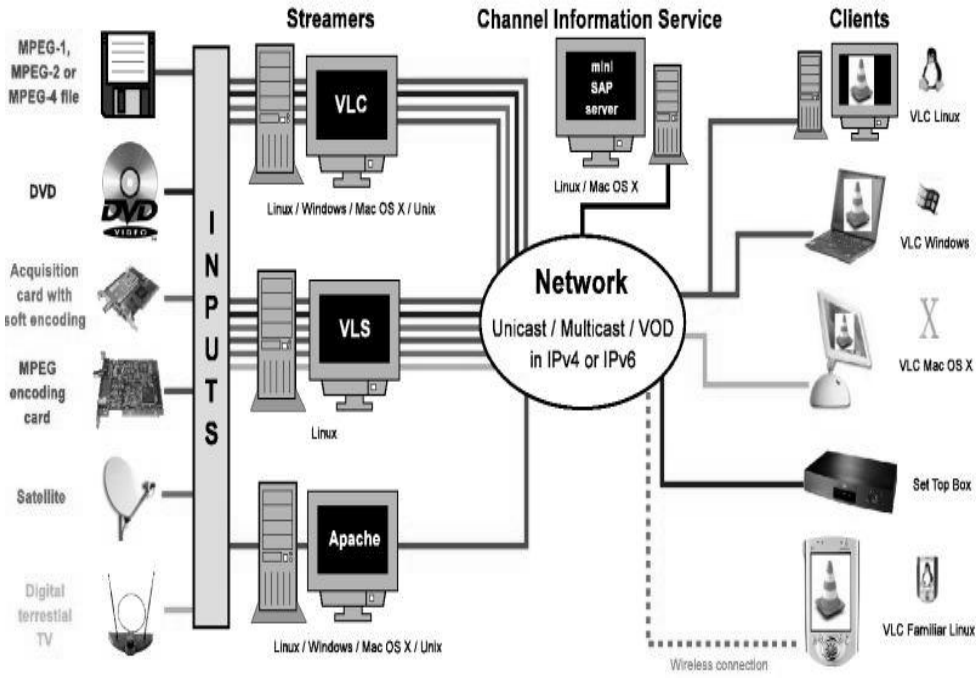

Figure 5.1: Global VideoLan Solution

6 Detailed Implementation

6.1 Rate-based Approach

6.1.1 Overview

This section describes the specifics of the implementation details for the components of the NIRS system. As shown in figure 6.1, the NIRS system is made up of a network profiling agent, an intrusion detection agent, and an active response agent using the pushback semantics for message propagation. The stages in the system operation are therefore divided to the network profiling stage, the intrusion detection stage and the active response stages.

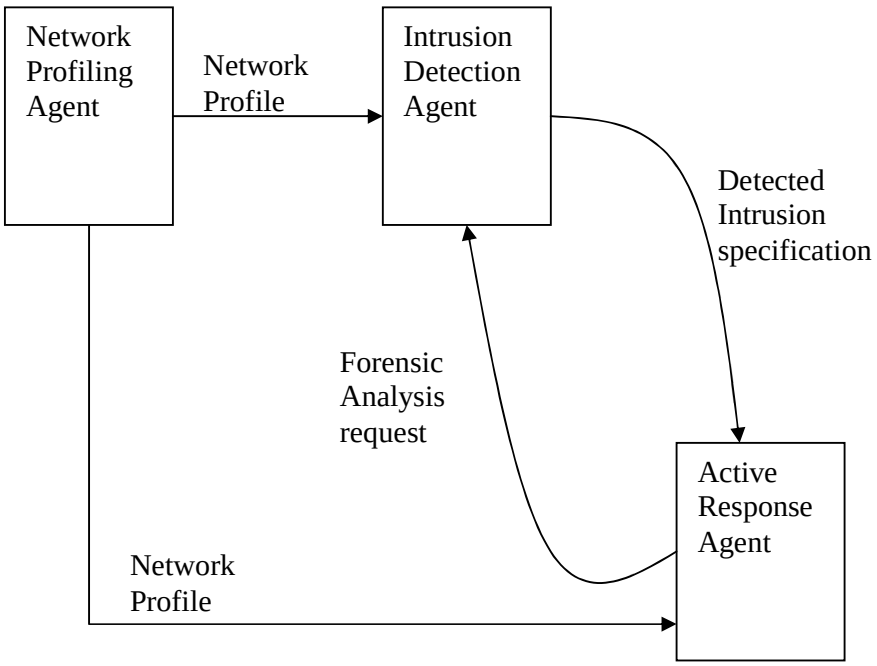

Figure 6.1: Network Profiling, Intrusion Detection and Active Response Agent

6.1.2 Network Profiling

In chapter four, the mathematical basis for intrusion detection was shown. This involved gathering network statistics against TCP, UDP and ICMP protocols by the profiling agent. These protocols are the most often used during network DoS attacks. This was done by setting up IP-accounting rules for the target or victim network and periodically reading the account data using the Linux iptables infrastructure described earlier in the report. For our tests in the project, the target network had address 192.168.1.0/24. In order to be able to capture all traffic information for this target network, the accounting function was placed at the ingress router serving this network segment. To set up the accounting tables for the target network against the TCP, UDP, and ICMP protocols, the following sequence of iptables rules were specified:

```
iptables -N ALLTCP
iptables -N ALLUDP
iptables -N ALLICMP
```

The iptables statements above would create new iptables chains required to store accounting information against the specified protocols.

Given that all packets destined for the target network and passing through the ingress router are passed through the *netfilter* OUTPUT or FORWARD chain of the ingress router, accounting information had to be gathered on these chains:

```
iptables -A FORWARD -p tcp -j ALLTCP
iptables -A OUTPUT -p tcp -j ALLTCP
iptables -A ALLTCP -d 192.168.1.0/24 -p tcp
```

The statements above tells the netfilter code to jump to the ALLTCP chain whenever a tcp protocol packet traverses the FORWARD or the OUTPUT chain of the router. The last statement isolates packets matching the separation criteria of either of the two statements above and with a destination 192.168.1.0/24 (the target network).

Similarly for the UDP and the ICMP protocols the following rules were used:

```
iptables -A FORWARD -p udp -j ALLUDP
iptables -A OUTPUT -p udp -j ALLUDP
iptables -A ALLUDP -d 192.168.1.0/24 -p udp

iptables -A FORWARD -p icmp -j ALLICMP
iptables -A OUTPUT -p icmp -j ALLICMP
iptables -A ALLICMP -d 192.168.1.0/24 -p icmp
```

To extract the accounting information from the netfilter system, the iptables list (**-L** option) command in the verbose (**-v**) will generate the packet and byte count that matches a rule specified to the netfilter kernel code. For example the command below shows accounting information for the ALLICMP chain specified above.

```
iptables –L ALLICMP–vnx
```

Output:

```
Chain ALLICMP (2 references)

 pkts bytes target  prot opt in out  source    destination
 1634 137256        icmp --  *   *   0.0.0.0/0 192.168.1.0/24
```

The output shows 1634 packets matching the rules have traversed the chain with a total size of 137256 bytes. To extract just this information, the command could be piped through grep, tr (translate character) and cut as follows:

```
D_IP=“192.168.1.0/24”
statICMP=`iptables -L ALLICMP -vnx -Z | grep "$D_IP" | tr -s '
' ':' | cut -d: -f2-3`
```

The result in the statICMP variable is packet count in field one and byte count in the second filed.

The **–Z** option in the iptables command zeroes the table for the next reading, such that accounting information of new packets that match all the rules specification only are read at the next sample.

This is done for all the protocols as shown below:

```
statTCP=`iptables -L ALLTCP -vnx -Z | grep "$D_IP" | tr -s ' '
':' | cut -d: -f2-3`
statUDP=`iptables -L ALLUDP -vnx -Z | grep "$D_IP" | tr -s ' '
':' | cut -d: -f2-3`
statICMP=`iptables -L ALLICMP -vnx -Z | grep "$D_IP" | tr -s '
' ':' | cut -d: -f2-3`

TCP_PKT_COUNT=`echo $statTCP | cut -d: -f1`
UDP_PKT_COUNT=`echo $statUDP | cut -d: -f1`
ICMP_PKT_COUNT=`echo $statICMP | cut -d: -f1`

TCP_BYTE_COUNT=`echo $statTCP | cut -d: -f2`
UDP_BYTE_COUNT=`echo $statUDP | cut -d: -f2`
ICMP_BYTE_COUNT=`echo $statICMP | cut -d: -f2`
```

These statements were put in a script, which executes continuously and prints out the results with a sequence number and the timestamp as shown below:

```
echo "tcp: $T $TCP_BYTE_COUNT $DATE $TCP_PKT_COUNT" | tr -s '
' "\t"
echo "udp: $T $UDP_BYTE_COUNT $DATE $UDP_PKT_COUNT" | tr -s '
' "\t"
 echo "icmp: $T $ICMP_BYTE_COUNT $DATE $ICMP_PKT_COUNT" | tr -
s ' ' "\t"
```

The output stream of this script was captured by the profiling agent, parsed and sent to its statistical analysis engine for processing. This is shown in Figure 6.2 below:

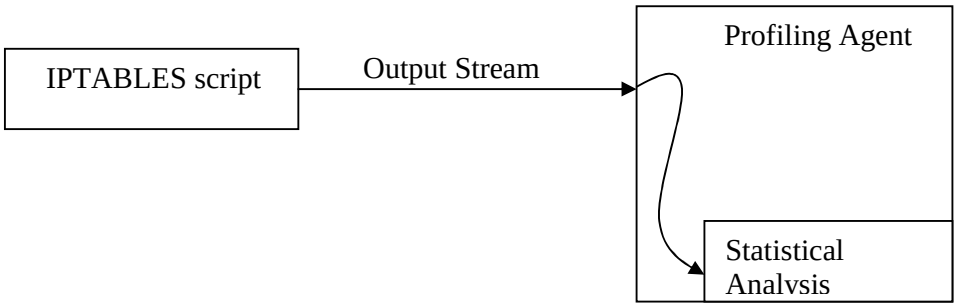

Figure 6.2: Operation of the Profiling Agent

The agent simply computes the mean and the standard deviation of the result over the profiling period. Given that the tests carried out in the in the project are to provide proof of concept for network profile based intrusion detection and response systems, the information fed to the profiling system has been greatly simplified. The categorization has been done at the protocol level, and the statistical analysis has been limited to the mean and the standard deviation of the training set. In a production system, it remains to be shown that further categorisation up to the protocol options level, isolation by source network, application layer information, and further statistical properties examination would greatly improve the quality of the intrusion detection system.

6.1.3 Intrusion Detection

Within the constraints of the standard deviation of the profiling data set from the mean, the intrusion detection agent is able to classify a network sample as an attack traffic or not. Several factors may make a legitimate traffic look suspicious, but

within the limits of information available, i.e. the mean and standard deviation, the level of confidence associated with a classification is affected by its value and the standard deviation of the training set. In set notation, this is shown diagrammatically below:

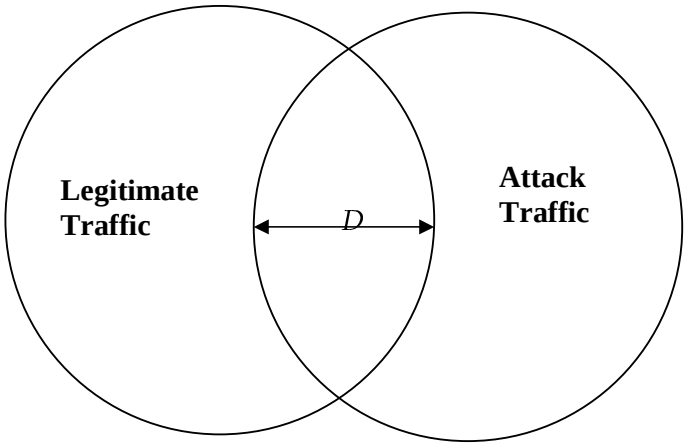

Figure 6.3: Normal and Attack Traffic Showing Region of Intersection

The area of intersection of the two sets shows the region of uncertainty for the intrusion detection system. We want to minimise this region as much as possible. The size **D** of the intersection region in the figure above is a factor of the variance of the data set. Further narrowing down the classification criteria of the training set could reduce this. However, within this constraint, the detection agent has to classify a network sample with an associated confidence level expressed as a probability **P(X)** given by the sigmoid function as described in Chapter four:

$$P(x) = \frac{1}{1 + e^{-ax}}$$

Intuitively, a = f (D) = f (σ) as previously shown in Chapter four. It remains to be shown the exact relationship between these variables. For the purpose of our tests, a was chosen to be 0.2.

Scaling our data sample to the region [-10,10] with our $[y_{min}, y_{max}] = [0,480000]$, then

$$x = \frac{y}{24000} - 10$$

Given that our maximum inbound link capacity is 60 kBps = 480 kbps.
With a = 0.2 and x = [-10,10] then P(x) = [0.1192,0.8808]

The intrusion detection system accepts the current sample, if it is less than the mean value from the profiling agent, it returns a P(x)=0 otherwise it passes the value through the sigmoid function and returns the associated probability. The detection engine also uses a parameter called confidence threshold **C**; whenever P(x) > C it sends an alert to the response system described shortly. A figure of the detection agent is shown below.

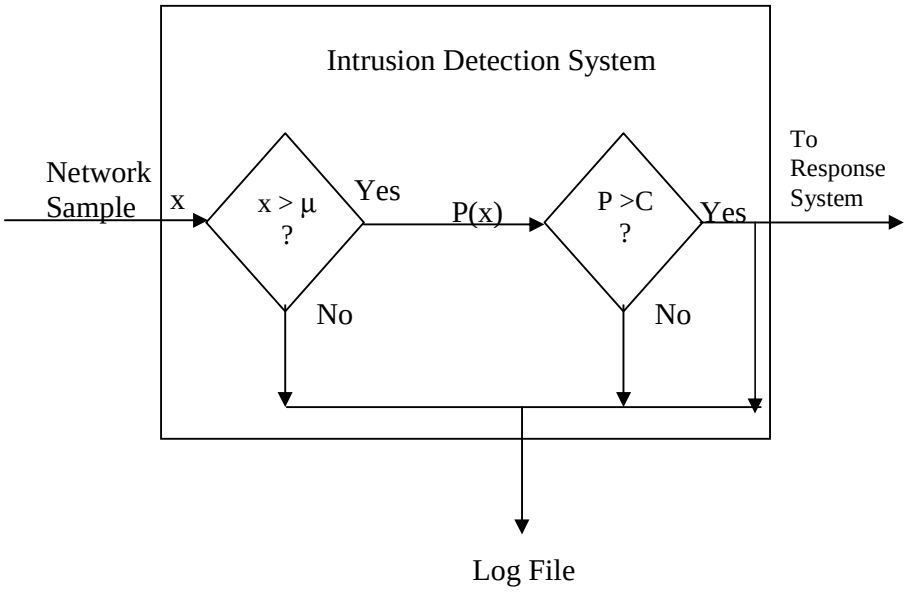

Figure 6.4: Schematic Of The Intrusion Detection System

6.2 Token Bucket Based Approach

The focus of this approach is not to develop a system different from that outlined in the previous chapter. Rather, we aimed to use a token bucket descriptor instead of aggregated rate as the basis for profiling, detection and response. We have implemented the detection system and experimented with the response system.

6.2.1 Detection System

6.2.1.1 System Design

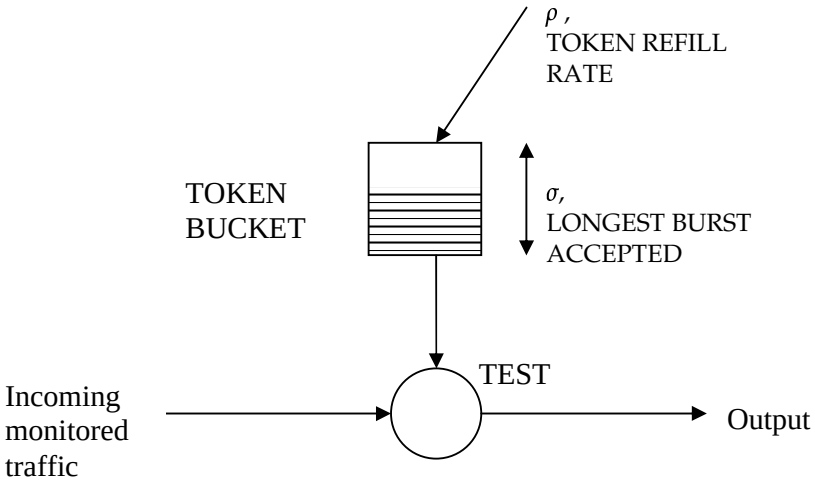

Figure 6.5: Token Bucket Policer

To detect if the incoming traffic conforms to the *normal* we had characterised, we implemented a token bucket policer with some modification. The design was depicted in Figure 6.5. The maximum number of tokens that the token bucket could hold was defined via the *bucket size* argument but no lower limit was defined. In fact,

we allowed the number of tokens in the bucket to be negative. The bucket is initialised to full and the token refill rates determined how many token would be refilled to the bucket per second. Packets from the traffic flow we would like to monitor entered our policer sequentially.

For each entering packet, it would need to 'consume' some number of tokens from the bucket. This 'consumption' was reflected by reducing the number of tokens in the token bucket by the number of tokens to be consumed by that packet. After this 'consumption', if the number of tokens in the bucket was non-negative, no alert would be raised as the flow conforms to the *normal* we had characterised. Otherwise, an alert would be raised with the number (negative) of tokens in the bucket also reported. If each packet "consumes" exactly one token regardless of its size, this conformance engine then checked for packet arrival rate conformance. On the other hand, if each packet "consumes" the same amount of token as its size, the engine then polices bandwidth usage.

6.2.1.2 Implementation

From our design, it seems the refill mechanism requires accurate asynchronous notifications such that tokens are refilled at the interval specified. One naïve approach could possibly be using SIGALRM to generate alarms at regular intervals to fill the bucket. However, such approach may not achieve very high accuracy given that Linux did not provide Real-Time guarantee. Even worse, the code would become unnecessarily complex to deal with potential problems of concurrency control.

In our implementation, we used inter-packet arrival time to determine the state of the token bucket. The advantage of such approach is that it does not require any external asynchronous notification. The insight of our implementation is that we only needed to know the state of the token bucket when a packet had arrived. Hence, we could use the event of packet arrival as a trigger. Here we present our argument.

Consider the token refill interval to be, *i,* and three consecutive packets to arrive at t_0, t_1, *and* t_2 respectively. If we assume that a refill has occurred at same moment as the first packet arrived. It is obvious that

$$t_1 - t_0 = n_1 * i + x_1 \quad \textbf{(6.1)}$$

where n_1 $(n_1 > 0)$ is the number of refilled should have happened between t_1, and t_0, and x_1 ($x_1 < i$) is the amount of time to next refill at t_1. As observed from Figure 6.5, we have,

$$(t_2 - t_1) = (n_2 + n_1) * i + x_2 - (n_1 * i + x_1)$$
$$(t_2 - t_1) = n_2 * i + x_2 - x_1$$
$$n_2 * i + x_2 = (t_2 - t_1) + x_1$$

where n_2 $(n_1 > 0)$ is the number of refilled should have happened between t_2, and t_1, and x_2 ($x_2 < i$) is the amount of time to next refill at t_2.

Therefore, the general form is:

$$n_j * i + x_j = (t_j - t_{j-1}) + x_{j-1} \quad \textbf{(6.2)}$$

where n_j $(n_1 > 0)$ is the number of refilled should have happened between t_j, and t_{j-1}, and x_j ($x_j < i$) is the amount of time to next refill at t_j. From our assumption, it was obvious that $x_o = 0$ by definition.

Assume that the number of tokens in the bucket immediately after consumption by the $(j+1)^{th}$ packet is N_{j+1}, which could be given by

$$N_{j+1} = N_j + n_{j+1} - (\text{number of token consumed by the } (j+1)^{th} \text{ packet}) \quad \textbf{(6.3)}$$

where N_o = maximum token bucket size.

From section 6.2, n_1 and x_1 are the quotient and the remainder of { $(t_1 - t_0) / i$ } respectively. Since $(t_j - t_{j-1})$ was in fact the inter-packet arrival time, and i is a known constant. The state of the token bucket after each packet has arrived solely depends on the inter-packet arrival time.

Since *iptables* does not provide any timing information on a per packet basis. We turned to packet capturing utility for such data. *Libpcap* is the *de facto* library for packet capturing under UNIX and Linux environment. For each packet captured, it provided information about the packets such as headers, length, etc. More importantly, each packet is time-stamped and thus allows us to compute the inter-packet arrival time. However, this timestamp reflected the time the kernel first saw the packet and it made no attempt to account for the time lag between when the ethernet interface removed the packet from the wire and when the kernel serviced the `new packet' interrupt [25].

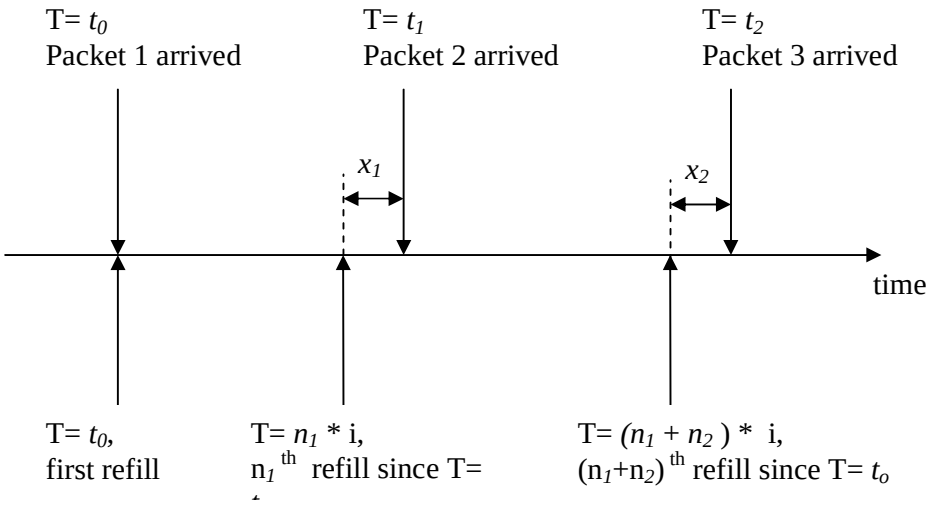

Figure 6.6: Inter-packet Arrival Time

Consider the Figure 6.6. The timestamp for packet #2 would be $t_0 + \Delta 1$ and the timestamp for packet #2 would be $t_1 + \Delta 2$. If we assumed that there was no large

variation in delay, when compared to td_1, during the capturing process, $\Delta 1 \approx \Delta 2$. We now had,

$$td_2 = (t_1 + \Delta 2) - (t_0 + \Delta 1) = (t_1 - t_0) + (\Delta 2 - \Delta 1) \approx t_1 - t_0 = td_1$$

As a result, the inter-packet capturing time would be a good approximation to the inter-packet arrival time and we established the rational to use *libpcap* in this detection system.

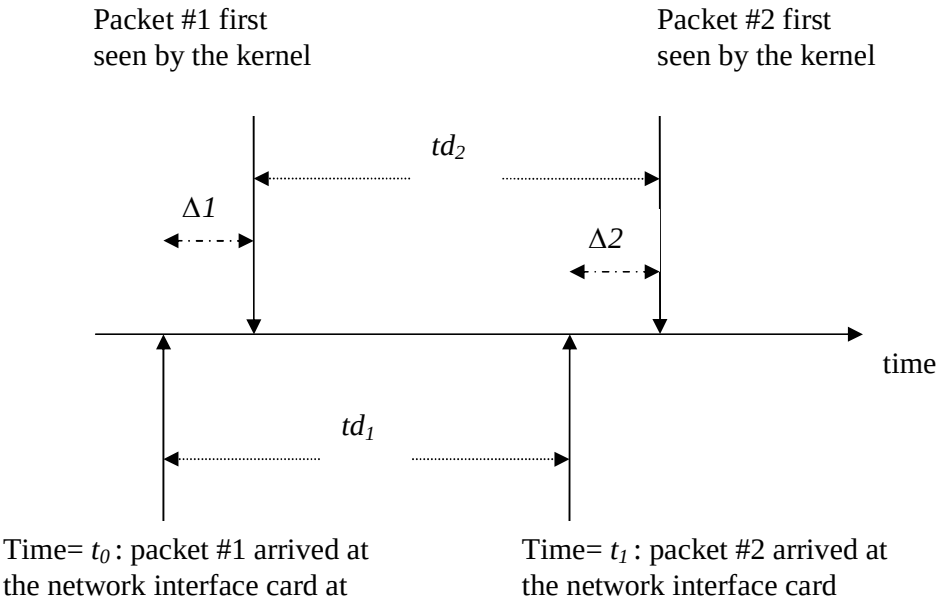

Figure 6.7: Packet Capture Time And Packet Arrival Time

We implemented, in C, our real-time token bucket-based detection as an extension module to Snort, the open-source intrusion detection system (IDS) built on top of *libpcap* library. There were several advantages in working within the framework of snort then using *libpcap* library directly.

First, *libpcap* is a powerful library, but to use it directly would require substantial expertise in UNIX/Linux system with regards to signals, low level devices, etc. The snort extension framework, called "*plugin"* in their documentation, hid all these complexity. Thus, we could concentrate in developing our detection module without having to spend weeks, if not months, to beef-up our UNIX system skills.

Secondly, the flexibility of snort was very helpful. Since we would like to build a set of network profile based on different header settings (indeed, protocol type could be considered as one form of header setting), we could leverage on the extensive header, and even payload, signature matching *plugins* to perform the task of protocol classification. In fact, by taking advantage of such capability, we needed to implement only a token bucket verifier *plugin*.

Third, snort is widely deployed thus it is a credible platform of choice.

6.2.1.3 Snort Plugin

The plugin we developed has three arguments: *token-refill rate*, *bucket size*, and *packet rate flag*. *Token-refill rate* configured the rate at which the token bucket would be refilled. It was expressed as token per second. *Bucket size* configured the maximum number of tokens that the token bucket could hold. *Packet rate flag* specified if the plugin should check for bandwidth or arrival rate attack. If this plugin was to check for bandwidth usage, it was assumed that each byte in a packet would consume one token. A sample snort rule that uses this plugin is as follow:

```
Udp any any -> 192.168.1.17/32 any ( token_bucket:"20000,
   50000,1"; )
```

This rule instructed snort to check if all UDP traffic destining for 192.168.1.17 would conform to the bandwidth usage of an average rate of 20000 bytes per second and the largest burst allowed is 50000 bytes.

Side notes

It was also important to point out that this *plugin* we developed can be used 'standalone' outside of the system we proposed. Currently, snort offered only signature-based intrusion detection. As pointed out in Chapter 3, such an approach was insufficient to counter the increasingly sophisticated and quickly evolving DOS attack. At the meantime, people started to demand to be able to describe attack, in snort, using signature in conjunction with rate in several newsgroup postings. Consider TCP sync-flooding. Currently snort was entirely ineffective such attack. It was because snort can only detect if such packets are matched on a packet-by-packet basis only. But the presence of any single TCP-sync packet did not constitute such an attack. It was the fact that such packets arrived at an abnormally high rate that constitute an attack. Using the token bucket plugin, system administrator could configure snort to raise an alert when a type of traffic was arriving at a token bucket rate higher then their historical value. Thus, they can then have a sound way to detect flooding-based attack without having to know the signature of the attack packet before hand.

6.3 Active Response System

The active response system is made up of a collaborating community of software agents. The master agent is the one closest to the resource being monitored. On detection of a DoS attack at the victim network, the intrusion detection system sends an alert to the response agent with information about the level of confidence it has on the attack and the information about the attack based on the two-tuple S={c, NET_INFO} described in chapter four.

Where NET_INFO contains network specific information like the offending protocol, IP addresses, interface etc required by the response system. The response system uses this information to react to the DoS attack. The response agent may still ask the

intrusion system to carry out further forensic analysis on the offending traffic to further isolate it from other legitimate traffic.

For our tests we used the netfilter rate-limiting functionality to throttle the data rate of the offending traffic. A limit is applied to the data rate of the offending protocol, and using the pushback messaging mechanism, the master agent sends signals to the upstream pushback messenger, which applies the rate-limiting function to traffic matching the profile specified by the master agent.

For the purpose of our test, the pushback message include the three-tuple {PROTOCOL, DESTINATION_IP, RATE} which the upstream agent interprets as: apply rate limiting to a maximum of RATE for all traffic destined for the downstream network with IP address DESTINATION_IP whose protocol is PROTOCOL. For example a message received by any pushback messenger {icmp,192.168.1.0/24,5} would mean rate-limit all icmp packets with destination IP in the range 192.168.1.0/24 to a maximum of 5 packets per second. In iptables parlance, this would be implemented as:

```
# create a new iptable chain to enforce maximum rate

CHAIN_NAME= ICMPLIMIT
DEST_IP=”192.168.1.0/24”
RATE=”5”

iptables -N $CHAIN_NAME 2>/dev/null

#flush the chain in case there was an existing rate
#specification

iptables -F $CHAIN_NAME

iptables -A $CHAIN_NAME -p icmp -d $DEST_IP -m limit --limit
$RATE/s -j ACCEPT
```

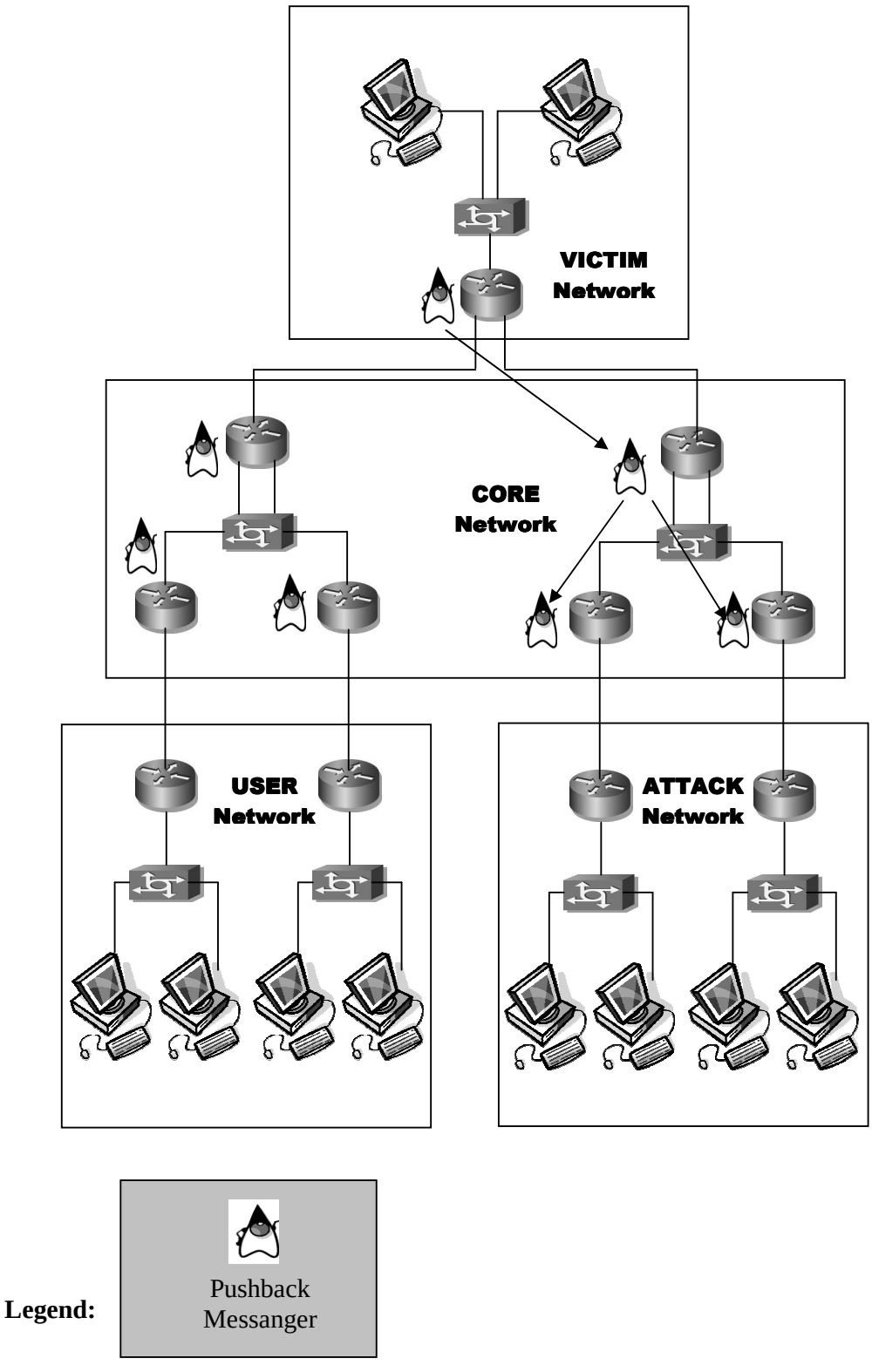

Figure 6.8: Pushback Message Propagation

This will apply the rate-limiting function to all ICMP traffic destined for the victim network. A further isolation parameter could be to use the source IP addresses of the offending traffic, but knowing that the source address may be spoofed, it is not a reliable method. Pushback message propagation is shown in Figure 6.5. The pushback messengers use lightweight UDP communication protocol.

We also experimented with the Linux traffic control utility, tc. It was developed as the control utility for Linux DIFFSERV (differentiated-service [26]) implementation and it interacted with the Linux kernel to setup network quality-of-service (QoS), such as traffic shaping (both ingress and egress) and traffic policing among others. In our setting, we used the traffic shaping feature.

In our experiment, we used tc to setup one “class-based” queue for each Ethernet interface. Queue here referred to queuing disciplines which were algorithms which control how packets are queued to the network interface card were treated. The concept of “class-based” queues would be demonstrated in Figure 6.8. Here we had a “class-based” queue which contains two “classes” and each “class” was associated with its own queuing discipline. “Filters” were used to match different packet property, such as header information”, and classified each of them into different “class”.

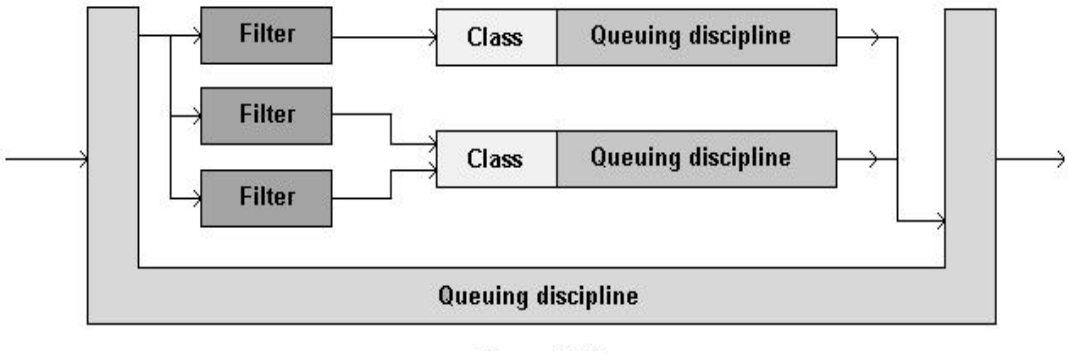

Figure 6.9: Logical Structure of Class-base Queue

We defined different “filters” to separate the traffic from the different traffic classes, as defined during the profiling phrase, into different classes in this “class-based” queue.

For each “class”, we then linked to each class a separate “token bucket filter” (TBF) queuing discipline to limit the bandwidth consumption to the level described by the token bucket parameters.

While tc performed excellently in bandwidth limiting, one disadvantage was that it did not support rate limiting. At least, we have not found out how it could be done with tc. Furthermore, the documentation and usage of the programme was rather poor and we had to rely on third-party documentation to understand the syntax and semantics of its “class-based” queues [27].

6.4 Packet Generator

We have three major requirements for the packet generator to be used. First, it must be simple to use. Secondly, we would like to be able to control the size of the packet it generates and the rate at which the packets are injected into the network at rate well below that is accepted by the virtual network infrastructure. Thirdly, it must be able to work with the Linux traffic control mechanism in order to constrain the traffic to be injected into the experimental network at rate configured.

There are many packet generators available on the internet, but we found most of them to be unsuitable for our purpose. While some of them, such as [28], have very complex configuration parameters due to their powerful nature, others, such as [29, 30] are designed to stress test network equipment such that they intend to send packets as fast as possible by interacting directly with the low level Ethernet device driver. It is not clear if packets generated this way would work with the queues

defined via the Linux traffic control mechanism and more importantly inside the UML environment.

Hence, we have developed our packet generator. It is a simple user-space C programme that sends fix-sized packet UDP to a given host at roughly a given time interval. There are three variants of the packet generators: one to generate “steady” traffic, one to generate “bursty” traffic, and one to generate packet as fast as possible.

6.4.1 Steady Traffic Generator

This generator takes the following arguments: destination host IP address and port number, UDP packet payload size (in bytes), inter-packet generation time (in seconds and microsecond). The minimum payload size accepted is four bytes (the rational will be clear) and the generation interval must be greater than zero. Upon start-up, it setups the UDP socket and then generates a buffer with size at least as big as that specified as the payload size and then. The first four bytes of the payload is used to accommodate a 32-bit integer. This integer, which is initialized as 0 and incremented by one for each packet sent, is used as a sequence number, in conjunction the packet receiver, to detect the number of packets dropped. It uses *sendto* system calls to inject packet into the network and then uses the *select* system to put the process into “sleep” for the duration specific before the next call to *sendto*. Since the system call actually takes time to complete, an inter-packet generation time of zero would certainly not be able to archive through this mechanism. In fact, we have developed another generator simply for this purpose.

6.4.2 Bursty Traffic Generator

One type of bursty traffic can be generated by a data source which transmits a sequence of packets for a particular period, also known as “busy” period, and then

becomes “silent”, a period which the source generate no network activity, for a relatively longer period. Typical example of a data source which exhibits such a traffic pattern is HTTP request generated by a user.

Our bursty generator attempts to emulate such data source and takes in the following argument: destination host IP address and port number, UDP packet payload size (in bytes), number of packets generated during the “busy” period, inter-packet generation time during “busy” period (in microsecond only), and the length of the “silent” period (in second and microsecond). While the length of the “silent” period is explicitly specified, the length of the “busy” is determined by the number of packet to be generated and the inter-packet generation time.

Similar to the steady one, payload size must be greater four bytes, and the time duration must be greater zero. Upon start-up, the generator setups the UDP socket and payload buffer similar to that of the steady generator. It then starts to generate packets with the inter-packet interval set to that specified for the “busy” period using *select* similar to the way described above. Indeed, the major difference is that after sending the number of packets during “busy” period as specified from the command line, the generator then enters a “silent” period. This is archived by using select to put the process into “sleep” for the duration of the “silent” period. Afterwards, it re-enters the “busy” period.

6.4.3 As-Fast-as-Possible Generator

We have attempted to develop a user-spec programme that generates UDP packets as fast as possible. The naïve implementation we have is to skimpily put *sendto* in a while loop which does not do much otherwise. However, we decided to abandon such approach because the UML environment cannot even keep up with packets generated through this approach.

6.4.4 Packet Receiver

This is a simple UDP daemon that accepts the packet sent to it by the generator. It checks the sequence number, inserted by the generator, of each packet against what it has received so far. Assuming no packet re-ordering has occurred, it then decides if any, and by roughly how many, packets have been lost during the transmission. This is particularly helpful when we engineered the link capacity of the experimental environment. This will be further discussed in the next chapter.

7 Test Environment

It is essential to have a testing environment to analyse and validate the proposition put forth in the previous chapters. The chapter is structured as follows: first is a description of the high-level design of the test environment, followed by details of how it was implemented and finally a discussion of the problems faced and their solution.

7.1 Design Overview

Figure 7.1 shows a detached section of the test environment to illustrate the network's multi-hop design consisting of routers and end-host. All end hosts are regular UML host but for 7the purpose of identification, we refer to host on the Victim subnet as Victim host, those on the attacking subnet are Attack host while others are called User host. The victim and user hosts were configured to be in different subnets.

The test scenario is laid out as follows: Under normal network operating conditions, good network traffic originates from the user subnet destined for a host in the Victim subnet as shown in the figure. Under attack condition, attack traffic is generated from a host in the attack subnet and also destined for the Victim subnet. This produces a situation where both the good and attack traffic compete for the same network resources. Thus emulating a Denial of Service (DoS) situation.

A tree like topology with a minimum tree depth of two was used for the core network with the *victim* located at the root of the tree and user /attack host are at the leaf nodes This tree-like topology was inspired by the one in [1] paper

There are several reasons for the choice of such a topology. First, in the view of the *victim* host or more specifically the server running on that host, the internet has a tree-

like topology with itself being the root. This is because it sees requests coming from different hosts flowing towards it. Such requests which may originates in different internet segment are all eventually aggregated into the one link connecting the *victim* host to the gateway router of that host (assuming the *victim* itself is not multi-homed). Second, such network allows us to study the effect of our response mechanism on different legitimate traffic on various level of aggregation with DoS attack traffic.

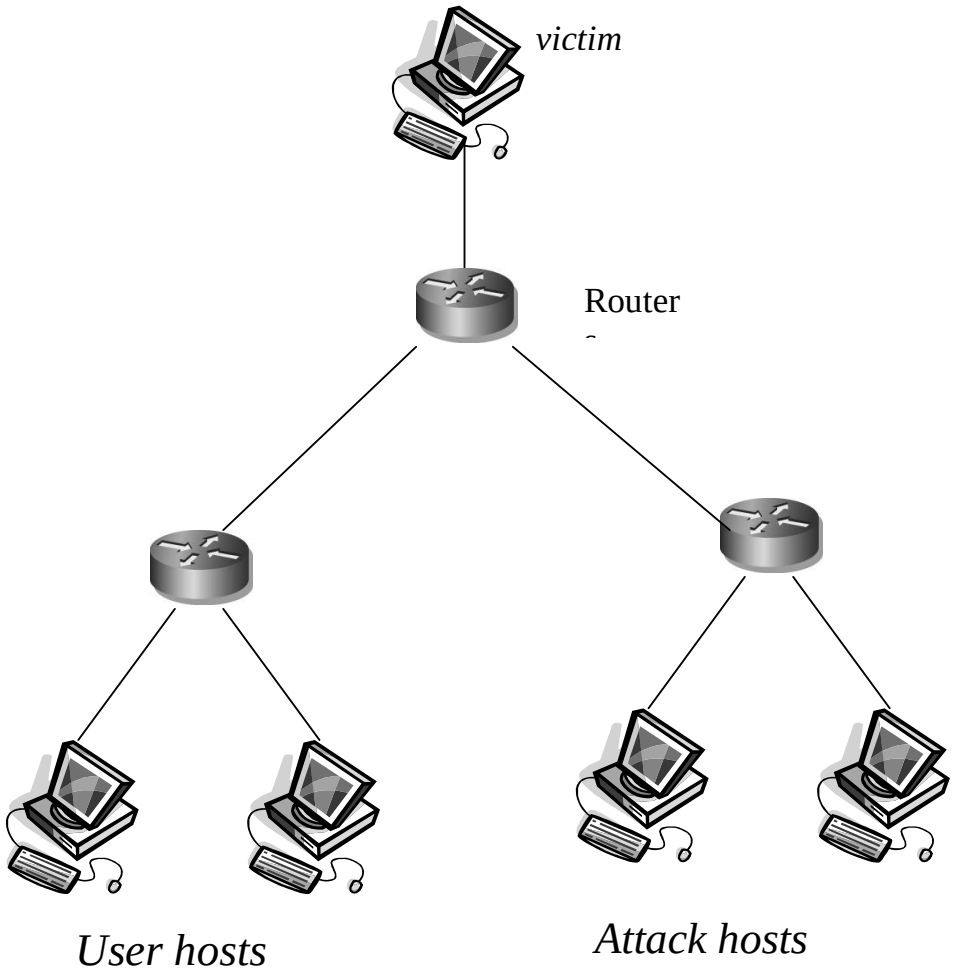

Figure 7.1: Design Of Our Test Network

7.2 Implementation

We would have loved to use real routers to implement our design. However, we had resource constrains. Indeed, we had only five Intel Pentium II – 233 Mhz PCs, one shared medium Ethernet hub and five straight ethernet cables to carry out our project. More importantly, the machines were only equipped with one network interface card and an extra card could not be added to any of the machines. This completely ruled out the possibility of changing how these machines were to be inter-connected to implement our design. Thus, the only solution available was to build an overlay network so that we could implement the topology designed.

The use of UML came as a natural choice to build such a network. Firstly, it allowed us to implement the network we had designed because each UML instance could be configured to have as many network interfaces as we desired. Secondly, since UML provided a full Linux environment, tools we developed for intrusion detection and response could be easily deployed in a real Linux environment without any significant changes. Thirdly, similar projects had been done with UML, e.g. the honeynet project [31]. So this technology was indeed a proven one.

7.2.1 Network Configurations

Fig. 7.2 shows the virtual network built using four ‘real’ Linux machines. This network consists of UML hosts configured as routers or end systems. These hosts were organized into three separate subnets (or networks) and a transit network such that each of these “components” resided in one “real” Linux machine. Each subnet was connected to the transit network via a virtual link with no direct connectivity between them. Inside each subnet, there was a gateway router which connected that subnet to the transit network and there were also a number of end systems. The gateway router and end systems were connected using *uml_switch*, a networking daemon that run inside “real” Linux machines to acts as an Ethernet switch to provide

connectivity among the UML instances running inside that machine. Similar to a real Ethernet switch, it can be configured to work in switch or hub mode.

For the subnets, we configured *uml_switch* to work as a hub. We named the transit network the **"core" network**. It consisted of only routers. These routers were connected using *uml_switch,* configured in switch mode, to emulate the nature of dedicated links among the routers. Finally, we connected the gateway router of each subnet and the "core" network through the physical LAN. We would like to point out that the choice of deploying this test network over more then one Linux machines was absolutely essential. From our live usage experience, the machines available for the project could not handle more than five UML instances running concurrently. As a result, we had no choice but to deploy the network over several physical machines. The topology is shown in figure 7.2 below

We named the three subnets: *attack network, user network*, and *victim network* respectively. The *victim network* was where the *victim* host, as described in section 7.1, resided. The *user network* is the network segment in the test environment where the legitimate user traffic is generated. The nomenclature used is indicative of the functionality in the test environment and is not related to its operational semantics. The operational semantics of the attack network is similar to that of the user network. But as its name indicates, DoS-type attack traffic originates from this network segment.

The IP addresses used for each node in this overlay network were selected from the non-routable address space (192.168.0.0/16) to prevent external traffic from interfering with our test environment and *vice-versa*. The victim, user and attack network segment in the test environment used classless inter-domain routing (CIDR) with a 28-bit network mask (255.255.255.240). This sets a theoretical limit of 14 hosts per subnet in each segment. The segments could have a theoretical limit of 14 subnets per router.

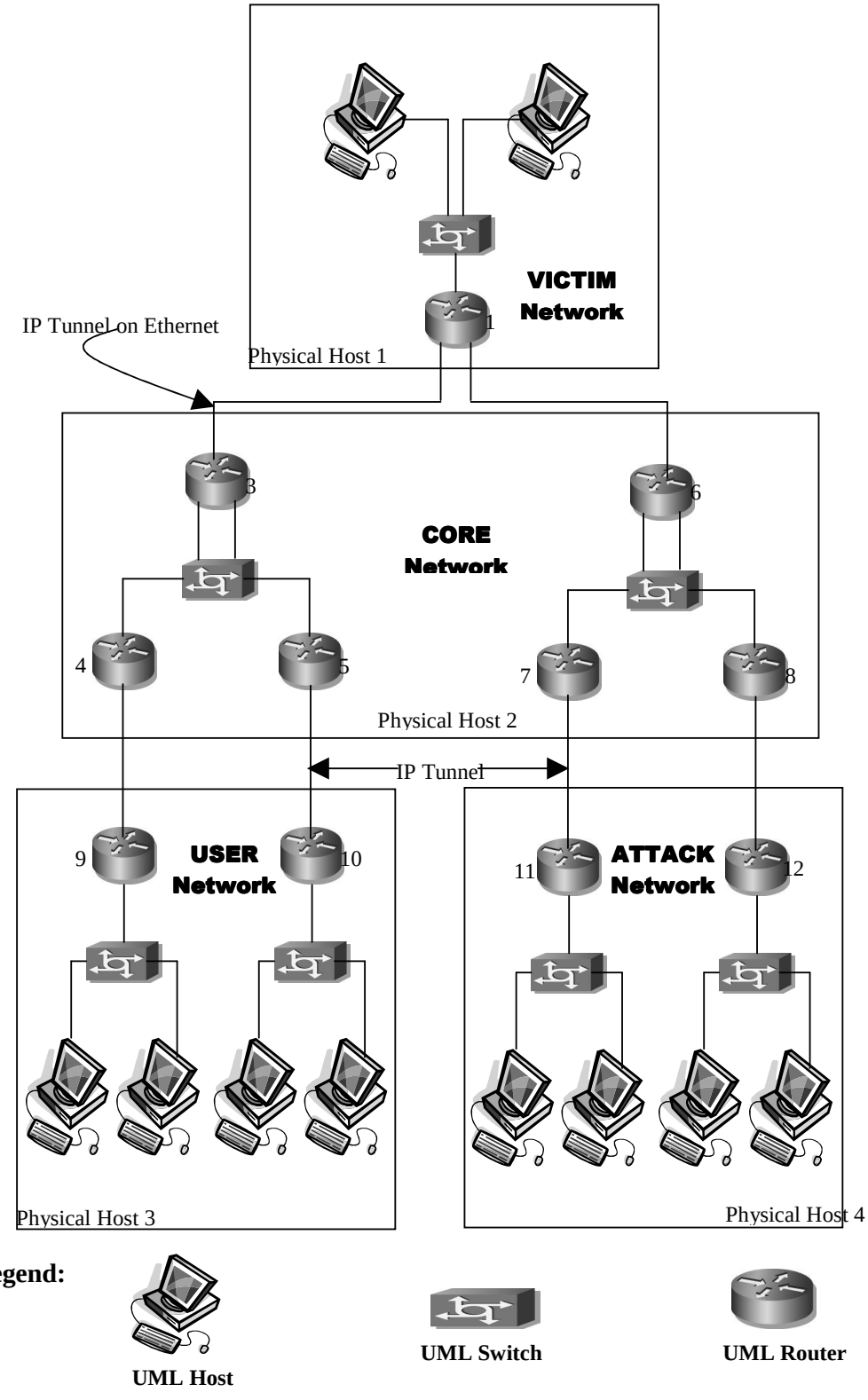

Figure 7.2: Topology Of The Virtual UML Network

The victim network had one router, the user network had two routers, and the attack network also had two routers. All outbound traffic from the subnets traversed the router through the core network to the destination and vice versa. The configuration was such that inter-subnet communication never traversed the physical host boundary to the physical network.

Given that the subnets and the "core" networks were connected using IP tunnelling over a shared-medium ethernet segment, which used CSMA-CD as the under-lying multiple-access scheme, care was needed in planning the data rate the virtual links would carry. As pointed out in [19], while load on an Ethernet increased, collisions became more common and thus increased the mean delay for each sender. In the extreme case if load tends to infinity, the actual throughput of the network drops to nearly zero.

Another rational for limiting the bandwidth of each cross-machine links was that these links were designed to be dedicated, such that traffic level on one link should not affect another. Since Ethernet, the underlying Layer-2 medium, does not support such notion, we had to "protect" each link from one another so that each of these links "behaved" similarly to a dedicated one.

To mitigate both problems, a maximum of 60% utilization was agreed on for the LAN. We limited the bandwidth of each cross-machine link to $1/n$ of the desired maximum offered load on the Ethernet, where n is the total number of cross-machine links (In this implementation, $n = 6$). Since Ethernet had a theoretical maximum of 10 Mbps data rate, this would amount to a total of 6 Mbps as the maximum desired offered load. Each of such links was capable of asymmetric data rate on the outbound and inbound links, but for the purpose of the experimental setup a symmetric data rate was assumed. Altogether, there were six cross-machine links each with symmetric data rate on the links; this left 6/12 Mbps for each link. This is equivalent to 62.5 Kbytes per second, and we further rounded it down to 60 Kbytes per second.

7.2.2 UML Configurations

UML being an implementation of the Linux kernel has similar networking semantics as a physical Linux box. During kernel compilation, networking options for routing, packet filtering, traffic control, IP tunnelling etc were enabled to facilitate the operation of different layers and components in the experimental setup that required the functionalities. For example, IP forwarding was required for UML hosts operating as router, and IP tunnelling was required for traffic isolation and proper routing of packets in the overlay network. These options were either compiled into the kernel or loaded on demand as modules during network setup. The following table shows the most important options and their contribution to satisfying the requirements for the overlay network specifications.

Table 7.1: Important UML Kernel Options For The Network

Networking Option	Kernel Option(s)	Description
UML Networking	CONFIG_UML_NET=y	Enable UML Networking
TUN/TAP Driver	CONFIG_UML_NET_TUNTAP=y CONFIG_TUN=y	TUN/TAP transport allows communication between UML host and the physical host
TCP/IP Networking	CONFIG_INET=y	Enable IP Networking
IP Forwarding	CONFIG_IP_ADVANCED_ROUTER=y	Enable UML host to operate as a router
IP Tunnelling	CONFIG_NET_IPIP=y	Support for IP in IP tunnelling
Packet Filtering	CONFIG_IP_NF_FILTER=y CONFIG_IP_NF_IPTABLES=y	Packet filtering and iptables support

All the traffic generated from the UML network were routed through an IP tunnel on the Ethernet LAN.

7.2.3 UML Environment

Since UML, by itself, is merely a Linux kernel running as a user-process in Linux, we needed to provide the filesystem in which the kernel will boot with and, in our case, the networking configurations of each instance.

We evaluated with several filesystem for UML available on the internet [32] and decided to use the one based on Debian *woody* distribution. The rational was that, even though RedHat based Linux distribution and its variants were widely used, those RedHat-derived file systems available from [32] did not suite our need. They were either too big in size, or lacked the tools we desperately needed, such as *make*, *perl*, *etc*. In fact, with the exception of the largest one, which is more then 600 MB in size, none of them came with *rpm*, the RedHat package management programme. This meant additional software could not be installed easily. On the contrary, the Debian-based filesystem was much very much smaller in size (25MB) and it had the Debian package management system installed. Hence, we were able to install all the additional software we required without much problem using *apt-get*, a Debian-specific utility to download and install software.

As laid out in section 7.2.1, we configured the UML instances as either end-hosts or routers. While they shared some common configuration, it was apparent that their networking configuration would be vastly different. In order to facilitate the deployment of this test network, we developed a set of scripts, which resides inside the UML filesystem. These scripts, during UML boot-up time via System V init process, would detect if it should be configured as a router or end-host and configure the Ethernet drivers, bring up the network interfaces, create the appropriate routing

table according to our topology, and, in the case of “border” router, setting up virtual IP tunnels appropriately.

While it was obvious to understand why we need to configure ethernet drivers, network interfaces, etc., it is less so with the need to setup virtual IP tunnels. Indeed, the need of such tunnels would be clear after we described the setup of the physical Linux boxes we had adopted.

7.2.4 ‘Physical’ Linux Host Environment

Even though a simple command was sufficient to start the booting of a single UML instance under Linux, it was no simple task to start a group of UMLs which were inter-connected in the way we devised. In our bid to make the UML environment as generic and flexible as possible, we put the intelligence of the topology in the *physical Linux box* environment instead. However, in doing so, we needed to “tell” each UML instance how to configure themselves when they are started. This was achieved by passing appropriate command line options to the UML instance such that it knows how it should configure itself and what values to use for the init scripts during boot-up as described in the previous section.

The requirement of the overlay network for the physical Linux host is that it must perform IP forwarding for the ‘private’ address space and address-resolution-protocol (ARP) request-reply on behalf of the UML instances running inside that particular host. Thus, it was necessary to configure the routing table of the physical host, as well as enable ARP proxy for those UML instance running that host.

However, this setting introduced a problem with the networking configuration between the gateway routers of the UML subnet and the *core* UML network. Consider the case of sending an IP packet from an end-host in the *user* subnet via the *core* network to another end-host, *v1*, in the *victim* subnet. The packet first got routed to the gateway router of the *user* subnet. Then, this gateway router would attempt to forward

this packet over the Ethernet to the corresponding router in the *core* network. In order to do so, the gateway router first forwarded this packet to the physical Linux box via the TUN/TAP device. At this point, the routing daemon of the physical Linux box would examine the header of the packet which would have the IP address of *v1* in this case. The routing table of the physical box would essentially contain a route to route this packet to the Ethernet device (eth0 in this case). As the physical box prepared to transmit this IP packet over eth0 device, it would have to discover the MAC address of *v1* through ARP. At this point, the physical host on which the *victim* subnet is running would certainly reply due to the ARP proxy configured as stated earlier. Thus, this packet would be delivered directly from the *user* subnet to the *victim* subnet without passing though the *core* network.

We solved this problem by using IP-in-IP (IP-n-IP) tunnel to connect gateway routers in each subnet to their peer router in the *core* network. Consider the above scenario again. When the IP packets arrived at the gateway router of the *user* network, the packet would be encapsulated into another IP packet destined for the peer router, *c1*, in the *core* network and this new IP-in-IP packet would then be forwarded to the physical host. When the routing daemon on the physical host examines this IP-in-IP packet, it would try to forward this packet to the address in the header of outer packet which is *c1*. This IP-in-IP packet would then be delivered to *c1* using the mechanism described in the last paragraph. On arrival at *c1*, this packet would be de-encapsulated and the original IP packet sent would now emerge. This packet would then be routed through the *core* network to the final destination *v1*.

We wrote a number of bash scripts to automate these tasks. These scripts, together with those residing inside the UML filesystem, were further brought together by a superset script that starts the whole network environment given a set of parameters that specify the topology of the desired network test environment. To make things simple, these scripts were written with the topology we described in mind, however, they were made fairly generic so that they could form a basis for creating a generic UML-based network test-bed for scientific experiments which require an emulated network environment.

7.3 System Evaluation

We have tested the setup with ping, our own packet generator and video streaming using VideoLAN. The result were satisfactory. We found no connectivity problem. However, we did observe several short-comings. For example, when an UML instance was transmitting too much traffic, the *uml_switch* seemed to be unable to keep up with the rate. This was mentioned in section 7.2.1. Another problem was observed with VideoLAN. The packet loss rate when streaming some selected video clip over this virtual network was very high. We attributed this problem to the fact that our equipment was not powerful enough for this particular task. As a control experiment, we repeated this experiment across the Linux machines themselves. Still, we observed a very large packet loss rate.

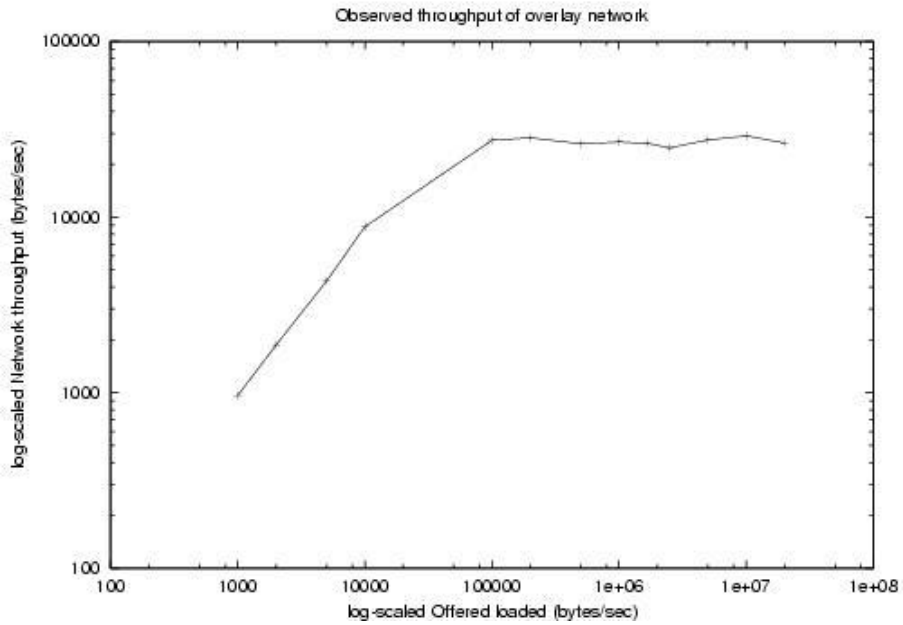

Figure 7.3: Observed Throughput Of The Network Against Offered Load By One Data Source

As laid out in section 7.2.1, we made provision to prevent a throughput collapse of the underlying Ethernet under high load offered by the UML end-hosts. Under stress test using our own packet generator to send packets from one host in the *user* network to another host in the *victim* network, we observed that the peak data rate remained constant even when the generator was generating packets at a much higher rate. This result, as shown in Fig 7.3, verified that the ethernet did not collapse even though under high offered load by an UML end-host.

We also verified all the cross-machine links were "protected" by running an instance of the packet generator in a host in both the *user* network and the *attack* network to send packets towards one host in the *victim* network.

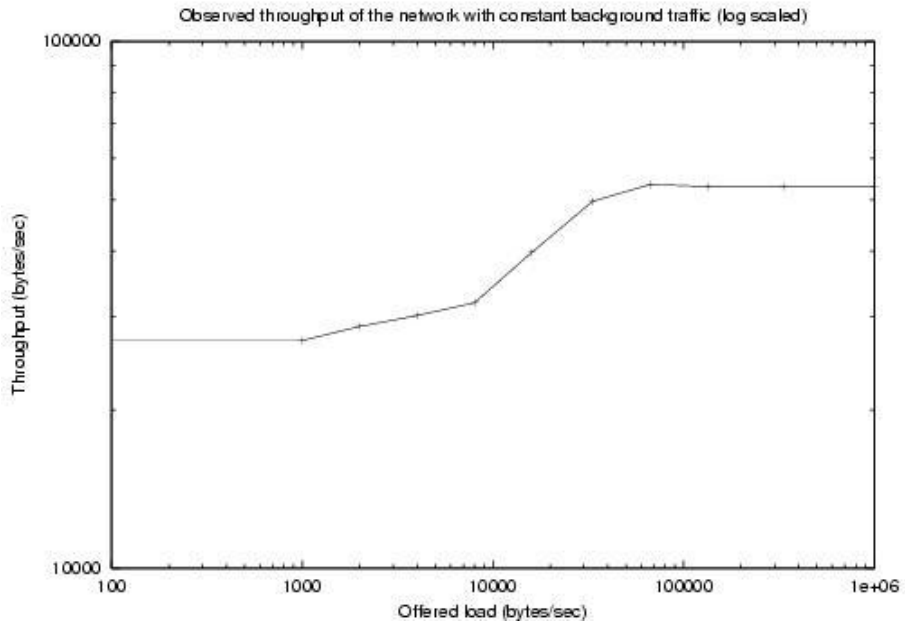

Figure 7.4: Observed Throughput Of The Network Against Offered Load With Constant Background Traffic

8 Measurement and Analysis

8.1 Background

In her paper *Testing Intrusion Detection Systems* [33] , Elizabeth B. Lennon identified a list of quantitative measures that relate to the performance accuracy of an intrusion detection system. An extract from her paper indicates the following measures:

Coverage

This measurement determines which attacks an IDS can detect under ideal conditions. For signature-based systems, this would simply consist of counting the number of signatures and mapping them to a standard naming scheme. For non-signature-based systems, one would need to determine which attacks out of the set of all known attacks could be detected by a particular methodology. The number of dimensions that make up each attack makes this measurement difficult.

Probability of False Alarms

This measurement determines the rate of false positives produced by an IDS in a given environment during a particular time frame. A false positive or false alarm is an alert caused by normal non-malicious background traffic. It is difficult to measure false alarms because an IDS may have a different false positive rate in each network environment, and there is no such thing as a standard network. Also important to IDS testing is the receiver operating characteristic (ROC) curve, which is an aggregate of the probability of false alarms and the probability of detection measurements. This curve summarizes the relationship between two of the most important IDS characteristics: false positive and detection probability.

Probability of Detection

This measurement determines the rate of attacks detected correctly by an IDS in a given environment during a particular time frame. The difficulty in measuring the detection rate is that the success of an IDS is largely dependent upon the set of attacks used during the test. Also, the probability of detection varies with the false positive rate, and an IDS can be configured or tuned to favour either the ability to detect attacks or to minimize false positives. One must be careful to use the same configuration during testing for false positives and hit rates.

Resistance to Attacks Directed at the IDS

Directed at the IDS. This measurement demonstrates how resistant an IDS is to an attacker's attempt to disrupt the correct operation of the IDS. One example is sending a large amount of non-attack traffic with volume exceeding the processing capability of the IDS. With too much traffic to process, an IDS may drop packets and be unable to detect attacks.

Ability to Handle High Bandwidth Traffic

This measurement demonstrates how well an IDS will function when presented with a large volume of traffic. Most network-based IDSs will begin to drop packets as the traffic volume increases, thereby causing the IDS to miss a percentage of the attacks. At a certain threshold, most IDSs will stop detecting any attacks.

8.2 Receiver Operating Characteristic

For an intrusion detection system, there is a relationship between the level of security that can be provided by the system and the attendant false alarm generated in

providing that level of security: i.e. the level of false alarm generated varies with the level of detection.

The trade-off between these two factors is governed by the alert threshold set for the intrusion detection system. By lowering the threshold, an administrator can discover more attacks, but will very likely have to content with more false alarms. Likewise, an administrator can raise the threshold to reduce false alarms, but this will also very likely cause the detection system to miss some attacks. Thus in evaluating an intrusion detection system, it is pertinent to know both the probability of detecting an attack and the probability of generating a false alarm. By knowing these two values and how they are affected by changes in the detection threshold, a Receiver Operating Characteristic (ROC) curve can be plotted. The ROC curve provides an administrator with requisite information that will enable him set a detection threshold that matches the level of security cum available effort required in his operating environment.

The ROC curve is an important characterisation used in the IDS testing community. The ROC curve for the NIRS system was generated to show the best operating point for the detection system. This led to the choice of the confidence threshold used during the full detection and response mode. The steps used in characterising the NIRS using ROC technique is outlined in the next sections.

8.3 Performance Objectives

While the parameters enumerated above could be used to provide quantitative measures that relate to the performance accuracy of an intrusion detection system, the focus of this work is to develop an active response system to DoS attacks. However, the effectiveness of the response system depends on the quality of the intrusion detection system. Therefore, the best operating condition for the detection system had to be determined using the ROC characterization technique.

8.4 Measurement Techniques

8.4.1 Measurement Conditions

In order to be able to take unbiased measurements of network data samples in the test environment, the physical network segment on which the virtual network was deployed was separated from the departmental network. This eliminated the possibilities of traffic from other sources interfering with our measurements. Furthermore, our test traffic could not stray outside our network segment or affect what other users are doing in other parts of the network

.

All network traffic used in the experiments were either generated using our custom made traffic generators or the ping utility. A “Good Traffic” source was defined, this was the traffic used to create the network profile. The “Attack Traffic” was defined also as an unruly traffic source, usually characterised by a high data rate.

8.4.2 Network Profiling

In order to characterise what is “good traffic” in the test environment, a traffic source of known behaviour was instantiated. The traffic source was either the packet generator we developed which is capable of sending steady rate or bursty UDP traffic to a specified destination, or the common ping utility. These accounted for two of the protocols we were investigating in the NIRS system. A stream of good traffic was deployed on the isolated network test bed from the user subnet to the victim subnet. The profile of this traffic stream was generated by the profiling agent on the ingress router of the victim network and serialized as a Java object to be used by the detection system during the detection mode. The following parameters constituted the traffic profile: average data rate and its standard deviation, average packet size and the average packet rate. The average data rate and its standard deviation were used as

inputs to the detection engine to be used in its decision logic during intrusion detection.

8.4.3 Confidence Level Plot

The data rates of packet streams on the overlay network are subject to variations due to the effect of network jitter, non-deterministic packet processing time in the overlay UML network and other sources, it was important to take care of these variations in our measurement and factor them into the selection of the appropriate confidence threshold of the detection engine. Given a network traffic sample, the intrusion detection system generates an output which is a confidence level that specifies the probability of that sample being an attack or not. The approach we adopted was to use the intrusion detection engine to sample the traffic at fixed time intervals. The observed sample values resulted in a range of probability values which is a map of confidence levels the detection system generated.

The result of a sampling session is shown in Figure 8.1 below. The information provided by this plot indicates the allowance to be given to the confidence threshold value due to the non-deterministic variation in good user traffic. This implies that good network traffic could occupy this range of values, and thus should not be signalled as an attack during the real network monitoring and intrusion response.

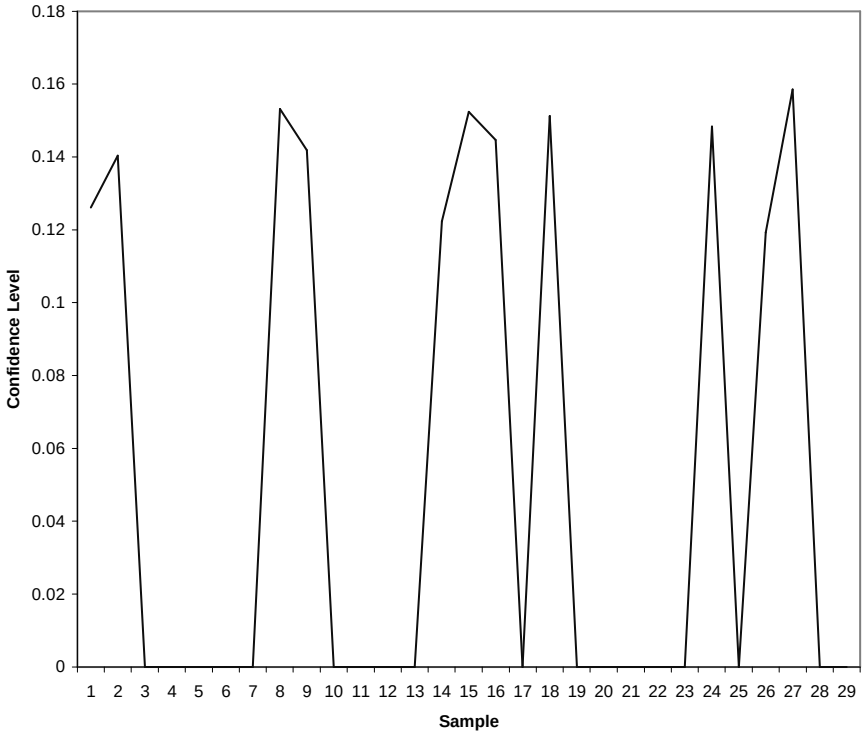

Figure 8.1: Confidence Level Plot For Good Traffic

Figure 8.2 is the confidence level plot of the intrusion detection engine when the traffic consisted of the good user traffic and attack traffic. With the overlap in the range of values in the good and the attack traffic plot, shown in Figure 8.3, care is required in the choice of the operating confidence threshold that would minimize the percentage of false positive while maximizing the detection rate. Figure 8.4 shows the ROC curve that determines the optimum operating point for the system.

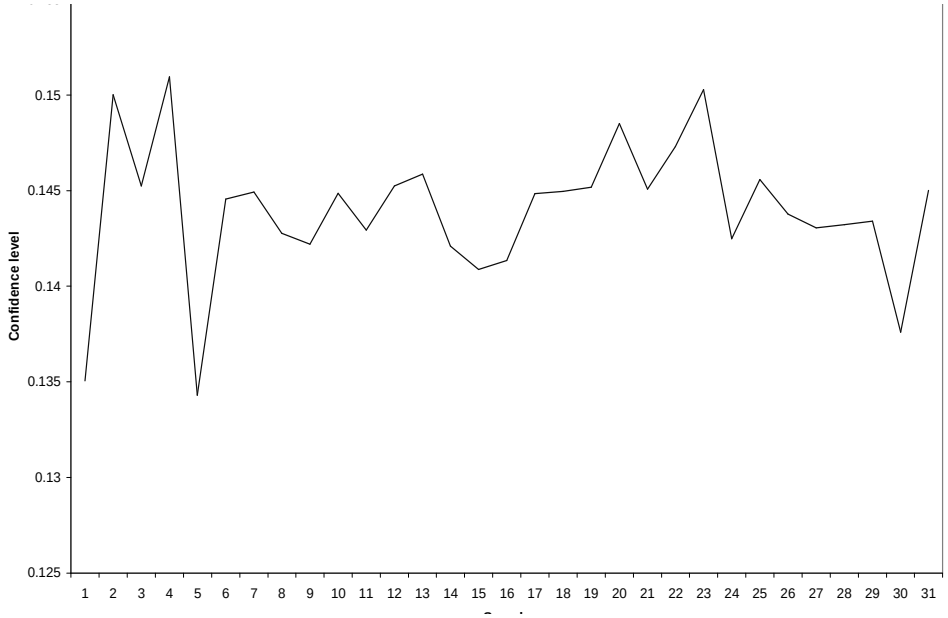

Figure 8.2: Confidence Level Plot With Attack Traffic

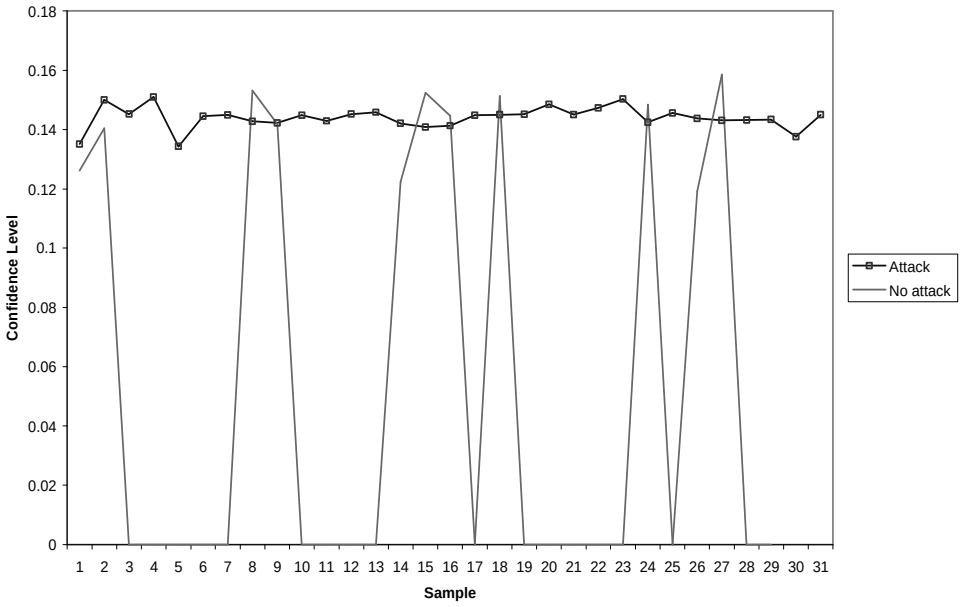

Figure 8.3: Confidence Level Plot Of Good And Attack Traffic Showing Overlaps

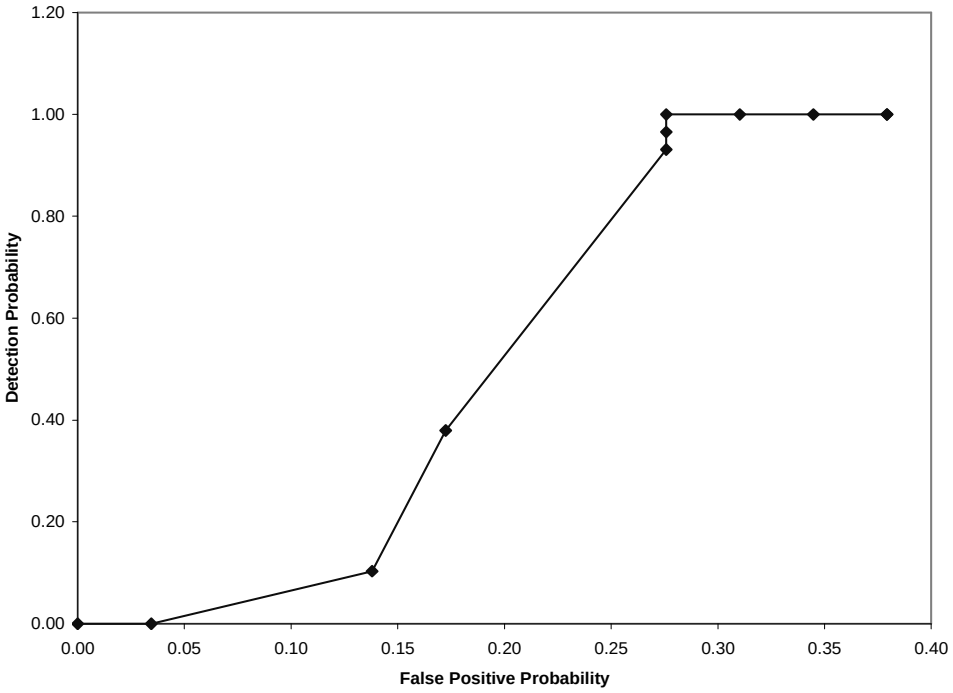

Figure 8.4: ROC Curve Showing Detection Probability With The Associated Level Of False Positives

Figure 8.1 shows the variation in confidence level generated by the intrusion detection system. Given that the parameters on which the IDS based its decision was the average rate and the associated standard deviation, it is difficult to be absolutely certain if a sample at any given period constitutes an attack or not. As a result, there is an associated confidence level the IDS generates with its output. The confidence level is zero if the sample is less than or equal to the average rate measured over the profiling period, otherwise it would be a value between zero and one. Obviously, samples with greater values than the average would generate a confidence level greater than zero, but should be small enough not to trigger the response system.

In our tests, the normal traffic confidence level were between [0.0000, 0.1586], while the attack traffic confidence plot shows confidence levels between [0.1343, 0.1510].

The overlap between the confidence ranges of the good and attack traffic makes the selection of the operating confidence level interesting.

As could be seen in Figure 8.3, the choice would affect the detection rate and an associated rate of false positives generated by the IDS. This is reflected by the ROC curve. The ROC curve shows that at 100% detection ratio, the minimum achievable probability of false positive is 0.28. That is, for 100% detection there would be a minimum of 28% false alarm rate. It could also be seen from the graph that there corresponds three different detection rate that correspond to the 28% false alarm rate. The meaning of this is that there are different confidence threshold levels that will result in the same false alarm rate, but one of them has the maximum detection probability.

For a system implementer expecting a 100% detection rate, the logical choice would be the confidence threshold that minimises the false positives. This ROC chart will generally help system implementers to select the best operating confidence threshold that suits their system.

8.5 Measurement Technique for Active Response

8.5.1 Active Response Mechanism

The effectiveness of the active response system depends on the quality and the accuracy of the detection system. An Analysis of the effectiveness of the detection system has been presented. This section describes the result of the active response system with the throughput variation of the good traffic before, during, and after the attack on the network. The response agent listens for alerts from the detection agent and responds using the pushback messengers that are located at strategic points within the network.

8.5.2 Measurement Technique

In order to observe the effect of the active response system on good and attack traffic, we set up a measurement script that monitors data rate of the good and the attack traffic at the attack target. To differentiate the two traffic types for the purpose of measurement, the attack was generated from a source different from the good traffic. As such, we could look at the source IP address to differentiate packets from the two streams. The amount of bytes of data measured from the different sources were logged for each network sample and plotted. This is shown in Figure 8.5. The good traffic source consisted of ICMP echo request packets, while the attack traffic was a ping flood. Another way to represent the effect of the ping flood on the good ping request traffic is to find the percentage of good traffic that arrives at the target over the sequence of network samples. This is shown in Figure 8.6.

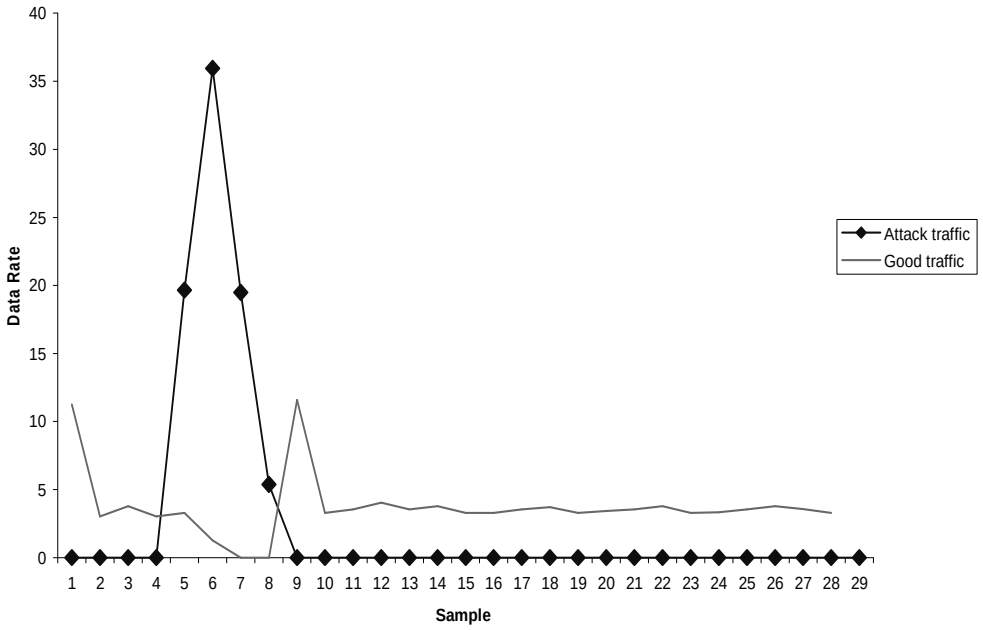

Figure 8.5: Data Rate Variation Of Good And Attack Traffic

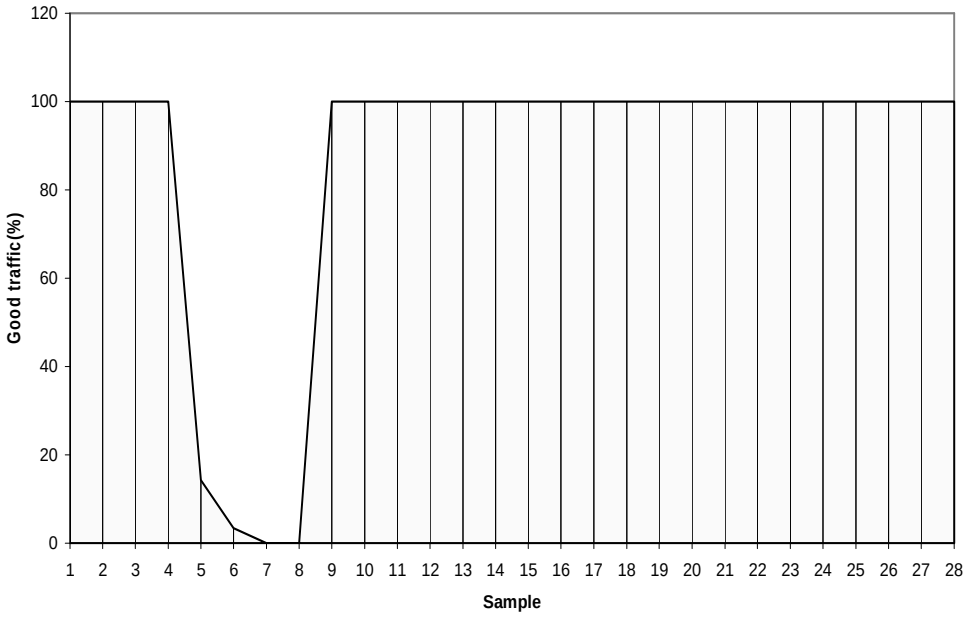

Figure 8.6: Percentage Of Good Traffic On The Network

Figure 8.5 showed the attack traffic flooding the network after the fourth sample. This had the effect of degrading the data rate of the normal ping request on the network. After the sixth sample the response system kicked in and applied rate-limiting function on the network against icmp traffic destined for the target network. The graph shows the attack could not get through as a result of the restriction on the traffic profile placed on in-coming icmp traffic which was on average of one packet per second with a maximum burst of 5 packets. This effect could also be seen on the good ping traffic in its burstiness that was constrained.

Figure 8.6 showed a plot of the percentage of good traffic arriving at the destination network. The percentage declined during the ping flood attack starting at sample four but came back up again after the active response.

9 Evaluation

This project started with the project title “mobile agent intrusion response system”. With many IDS developed focusing on the detection of DoS attack on a network, our group wanted to enhance the IDS by focusing on the development of a response system as critical add-ons for current IDS. The idea was to use mobile agent for intrusion detection and response in active networks. However as the project progressed and more materials were gathered on previous and current research on IDS, coupled with lack of certain functionalities in Snort, our IDS of choice, the group shifted its direction towards developing its own IDS together with a new response mechanism using the pushback mechanism.

There are several kinds of network attacks and it is impossible for one system to be able to detect everything. One of the popular attacks, DoS attack is a very critical threat and has been a big thorn to the flesh of many network administrators. The attacker uses DoS to abuse network bandwidth resulting in denial of users to network services. With the time constraints that the group had, and its security importance we decided to concentrate on DoS attack.

Many of the limitation in many current IDS is because they are misuse-based systems. This is true with Snort which detects intrusion by matching signature attacks that have been programmed into it. Also, we found out that Snort does not do rate detection or rate limiting on network traffic. This made it unsuitable for the purpose of rate detection which was a key feature in our approach to using rate detection and limiting as our response mechanism.

We have been able to develop two IDS using anomaly detection approach. One was a rate-based IDS that detected intrusion by analysing deviations from the normal traffic profile. Another IDS used Token Bucket implementation to detect intrusion. The project tackled the DoS attack by treating them as traffic violation problem.

We started with the Token Bucket implementation as our IDS. Token Bucket worked together with Snort to create an IDS that detected intrusions given a traffic characterisation derived from a linear bound arrival process algorithm. Token Bucket operated with a burst rate and traffic arrival rate. In the duration of our project, significant time was spent in getting the leaky bucket parameters. However, given the time constraint that we had, the group decided to continue the project with a different approach. Hence, the work on rate-based IDS.

After the two IDS works were finished, the group started working on the response aspect of the NIRS and decided to use the pushback mechanism. We found many benefits to using this mechanism. One of them was that pushback would be most effective when there are routers at a position near to the target from where most of the offending traffic will be arriving from. This suited our small test environment very well.

Out of the four months allocated for this project, the group spent a month to figure out the UML Networking and having it stabilized in our environment. UML is a new subject to all four of the group members and we spent a great deal of our time trying to understand the workings of it. We relished the opportunity to work on a new area and in the end managed to overcome the challenges presented to us.

In the end, we believe that the Network Intrusion Detection and Response System that we have developed is a very useful project. When used together with a pattern matching IDS, it can become a more powerful tool to combat DoS attack.

9.1 Project Management

Group 4 was formed by four DCNDS students whose interests lie in the computer security area. We are all from different cultural and education backgrounds which ensures a good diversity of technical perspectives and approaches to problem-solving.

From the very beginning, all four of us understood the aim of this project and decided to divide the tasks equally among all four of us depending of the strengths and the weaknesses of the individuals. We would take advantage of the strengths we had and assists others with their weaknesses. At the start, the group agreed to adopt a flat team structure in which one person will champion an area of responsibility in the project but have others involve in every aspect of the project.

Table 9.1: Group Structure

Adedayo Adetoye	Technical
Andy Choi	Organization
Marina Md. Arshad	Documentation
Olufemi Soretire	External Liaisons

The group worked in the department's lab everyday. The day started with daily meetings at 11:00 am. Also, we met weekly with our supervisor Dr. Steve Hailes to inform him of our progress and get input from him. It is given that with four persons in a group with their own ideas and way of doing things, there would be small conflicts and disagreement. During our lengthy discussions, each and everyone had the right to voice out their ideas and concern but in the end, everyone will agree upon one idea.

Because we work together all the time and everyday in the lab, we managed to resolve all setbacks quickly without major glitches. Integration and final testing was organised and done together and the group managed to finish the work satisfactorily.

10 Future Work

To build on the success we had so far, we had already identified several directions that would warrant further investigation.

10.1 Rate-based Detection

First, we believe that we would improve the granularity during the profiling phrase. Our current implementation *polled* for information roughly every 10 seconds. Consider a 10 Mbps link. At peak capacity, about 12.5 Mbytes could have passed through the network during that time interval. The aggregation here is quite large. Also, timing in Linux/Unix was not very accurate.

To improve this aspect, we could look into using some form of asynchronous notification of packet arrival. Perhaps, we could take advantage of the *iptables* extension framework to do so. Furthermore, any burst occurred during that interval would not be characterised towards to traffic profile because it had been aggregated into the aggregated rate. Already, we had started to study the possibility of replacing aggregated rate with a LBAP/token bucket descriptor.

However, we could look even further for other more advanced, and probably experimental, form of traffic descriptor. Another dimension of granularity we would improve would be that of different traffic classes. Currently, we profile network traffic at protocol level. It might be useful, for some protocol, to further classify traffic within that protocol. For example, it would be useful to isolate TCP-SYN, from other TCP traffic to better detect TCP-SYN flood.

Secondly, it is not unusual for network traffic to fluctuate with time of day or week due to human habits and procedures. We could improve our detection system by

using time-variant parameters to reflect such fluctuation. Using principles of artificial intelligence, the profiling agent could learn about such time dependent variables and factor it in to the detection system for better resolution of its detection mechanism.

10.2 Response

As we improve the granularity of the detection system, the response components must also keep up to the improvements made. Given our response mechanism was based on message between routers, there would be security implications. For example, how would routers authenticate with one another? How would the routers check the integrity of the messages it received? IP Security (IPSec) might provide some solution, but it also introduced the problem of implementing a secure domain for every router. Furthermore, how should the certificates be configured? Should that be on a host-by-host basis or interface-by-interface basis? If security issues fully addressed, this automatic response mechanism, we proposed, would in fact be a big security liability. Quite the opposite we would like to see.

The automated response system involved changes the way traffic flowed in the network through automated provisioning. While this would be helpful in defence against DOS attack in our experimental environment, we need to further study if such scheme could be scaled up. Equally importantly is how the traffic dynamics would be affected by such changes. Would the routers attempt the re-route the traffic along another path so that our “defence” for network was, in fact, circumcised by the network itself?

Another issue would should this automated response be managed. In order to be deployed in a live system, administrator would like to be able to have some control over this scheme. For example, he may decide that some routers not to be part of the automatic response infrastructure. A management plane was lacking.

10.3 Token Bucket

First, we need to implement the profiler and linked it with the components we have developed. Then, we need to test it under the same environment to observe if it would offer any advantage to the aggregated rate approach. We could consider implementing the profiler within the snort framework again. Furthermore, as we pointed out in chapter 4, token bucket descriptor was not capable to describe very bursty traffic. Similar to our proposal for rate-based profiler, we could adopt some AI techniques here to characterise network traffic as several profiles where the burst within each profile was relatively small.

Since the token bucket verifier, implemented as a snort plugin, involved floating pointing mathematics, we would study the effect of such rather complicated logics on performance. We would like to measure the number of CPU cycles required to process each packet. It would also be our interest to benchmark what the maximum incoming packet this implementation could cope with.

As we pointed in chapter 6, tc currently does not provide rate limiting functionality, we should explore how rate limiting, based on token bucket descriptor, could be archived.

11 Conclusion

Traffic profile-based approach could be used as a defensive measure against DoS-type network attack. In this report, we have proposed two different candidates traffic characterisation techniques: aggregate rate, and token bucket descriptor. We successfully implemented an IDS based on the aggregated rate, and made some progress towards supporting token bucket descriptor. This system was tested and validated in an overlay network which we built using UML. While a lot of remains to be investigated, the results from our analysis pointed us towards a very promising direction.

The project result aside, we gained enormous experience in the internal work of the networking subsystem on Linux systems, as well as different aspects of network intrusions detection. Finally, we were able to applied we have learned from the different course modules in the course of the project.

References

[1] S. M. Bellovin J. Ioannidis. *Implementing pushback: Router-based defense DDoS attacks.* in *Network and Distributed System Security Symposium.* February 2002. Catamaran Resort Hotel San Diego, California.

[2] J.P. Anderson, *Computer Security Threat Monitoring and Surveillance*. April 1980, James P. Anderson Co.

[3] D.E. Denning, *An Intrusion-detection Model.* IEEE Transaction on Software Engineering, February 1987. **SE-13**(2): p. 222-232.

[4] D. Duggan T. Draelos, M. Collins and D. Wunsch. *Adaptive Critic Design for Host-Based Intrusion Detection.* in *2002 International Joint Conference on Neural Networks.* May 2002.

[5] F. Roli G. Giacinto, and L. Didaci, *Fusion of Multiple Classifiers for Intrusion Detection in Computer Networks.*

[6] T. Hong D. Joo, I. Han, *The neural network models for IDS based on the asymmetric costs of false negative errors and false positive errors.* Expert Systems with Applications, 2003: p. 69-75.

[7] J. Haile J. Larsen, *Understanding IDS Active Response Mechanisms*, SecurityFocus, January 29, 2002, http://www.securityfocus.com/infocus/1540

[8] *SPADE,* Silicon Defense, http://www.silicondefense.com/software/spice

[9] M. Bendre R. Sekar, D. Dhurjati, P. Bollineni. *A Fast Automaton-based Method for Detecting Anomalous Program Behaviors*. in *2001 IEEE Symposium on Security and Privacy*.

[10] T.F. Lunt D. Anderson, H.S. Javitz, A.Tamaru, and A.Valdes, *Detecting unusual program behavior using the statistical component of the Next-generation Intrusion Detection Expert System (NIDES),* in *Technical Report SRI-CSL-95-06*. May 1995.

[11] P.K. Chan M. Mahoney, *PHAD: Packet-Header Anomaly Detection for Identifying Hostile Network Traffic*. 2001-04, Florida Tech. p. 376-385.

[12] P.K. Chan M. Mahoney. *Learning Nonstationary Models of Normal Network Traffic for Detecting Novel Attacks*. in *Proceeding SIGKDD*. 2002. Edmonton, Alberta, Canada.

[13] P.K. Chan M. Mahoney, *Learning Models of Network Traffic for Detecting Novel Attacks*. 2002, Florida Tech.

[14] A. Gupta R. Sekar, J. Frullo, T. Shanbhad, A. Tiwari, H. Yang, and S. Zhou. *Specification-based Anomaly Detection: A New Approach for Detecting Network Intrusions*. in *Proceedings of the ACM Conference on Computer and Communications Security 2002*. November 2002.

[15] S. J. Stolfo W. Lee, and K. W. Mok. *Mining in a data-flow environment: Experience in network intrusion detection.* in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD-99)*. 1999.

[16] S. J. Stolfo W. Lee, *A framework for constructing features and models for Intrusion Detection System.*

[17] A. Christie J. Allen, W. Fithen, et. al., *State of the Practice of Intrusion Detection Technologies*, in *CMU/SEI-99-TR-028*. 1999.

[18] K. Das, *Protocol Anomaly Detection for Network-based Intrusion Detection*, in *GSEC Practical Assignment Version 1.2f*.

[19] Keshav, *Engineering Computer Networks*. 1999.

[20] Turner, *New Direction in Communications.* IEEE Communication Magazine, October 1986. **25**(10): p. 8015.

[21] P. Varaiya S. Low, *A simple theory of traffic and resource allocation in ATM.* In the proceedings of GLOBECOM '91, December 1991. **volume 3**: p. 1633-1637.

[22] A. Varma C. Tryfonas, S. Varma. *Efficient Algorithms for Exact Computation of the Loss Curve of Video Source*. in *Proceedings of Packet Video'99*. 1999.

[23] *ping(8) - Linux man page*, http://www.die.net/doc/linux/man/man8/ping8.html

[24] *Differentiatied Service on Linux*, http://diffserv.sourceforge.net/

[25] *tcpdump - dump traffic on a network man page*, http://www.tcpdump.org/tcpdump_man.html

[26] R. Sahita K. Chan, S. Hahn, K. McCloghrie, *Differentiatied Services Quality of Service Policy Information Base*. March 2003.

[27] Leonardo Balliache, *Practical QoS*, http://opalsoft.net/qos/

[28] Roelof Jonkman, *NetSpec*, http://ittc.ku.edu/netspec/

[29] NetPIPE, *A Network Protocol Independent Performance Evaluator*, http://www.scl.ameslab.gov/netpipe/

[30] R. Jones, *Netperf*, http://www.netperf.org/netperf/NetperfPage.html

[31] *Honeynet Project*, http://project.honeynet.org/

[32] UML, *The User-mode Linux Kernel Home Page*, http://user-mode-linux.sourceforge.net/

[33] *Testing Intrusion Dectection Systems*, E.B. Lennon, Editor. July 24th 2003, Information Technology Laboratory, National Institute of Standards and Technology.

APPENDIX A: Compiling the User Mode Linux kernel and modules

Compiling the Kernel

Compiling the user mode kernel is just like compiling any other kernel. Let's go through the steps, using 2.4.0-prerelease (current as of this writing) as an example:

1. Download the latest UML patch from the download page (http://user-mode-linux.sourceforge.net/dl-sf.html). In this example, the file is uml-patch-2.4.0-prerelease.bz2.
2. Download the matching kernel from your favourite kernel mirror, such as: http://ftp.ca.kernel.org/linux/kernel/ http://ftp.ca.kernel.org/linux/kernel/ .
3. Make a directory and unpack the kernel into it.

   host% mkdir ~/uml

   host% cd ~/uml

   host% tar -xjvf linux-2.4.0-prerelease.tar.bz2

4. Apply the patch using

   host% cd ~/uml/linux

   host% bzcat uml-patch-2.4.0-prerelease.bz2 | patch -p1

5. Run your favorite config; `make xconfig ARCH=um' is the most convenient. `make config ARCH=um' and 'make menuconfig ARCH=um' will work as well. The defaults will give you a useful kernel. If you want to change something, go ahead, it probably won't hurt anything.

   Note: If the host is configured with a 2G/2G address space split rather than the usual 3G/1G split, then the packaged UML binaries will not run. They will immediately segfault.

   Finish with `make linux ARCH=um': the result is a file called `linux' in the top directory of your source tree. You may notice that the final binary is pretty large (many 10's of megabytes for a debuggable UML). This is almost entirely symbol information. The actual binary is comparable in size to a native kernel. You can run that huge binary,

and only the actual code and data will be loaded into memory, so the symbols only consume disk space unless you are running UML under gdb. You can strip UML:

host% strip linux

to see the true size of the UML kernel.

Make sure that you don't build this kernel in /usr/src/linux. On some distributions, /usr/include/asm is a link into this pool. The user-mode build changes the other end of that link, and things that include <asm/anything.h> stop compiling.

The sources are also available from cvs at the project's cvs page , which has directions on getting the sources. You can also browse the CVS pool from there.

If you get the CVS sources, you will have to check them out into an empty directory. You will then have to copy each file into the corresponding directory in the appropriate kernel pool.

If you don't have the latest kernel pool, you can get the corresponding user-mode sources with

```
host% cvs co -r v_2_3_x linux
```

where 'x' is the version in your pool. Note that you will not get the bug fixes and enhancements that have gone into subsequent releases.

If you build your own kernel, and want to boot it from one of the filesystems distributed from this site, then, in nearly all cases, devfs must be compiled into the kernel and mounted at boot time. The exception is the tomsrtbt filesystem. For this, devfs must either not be in the kernel at all, or "devfs=nomount" must be on the kernel command line. Any disagreement between the kernel and the filesystem being booted about whether devfs is being used will result in the boot getting no further than single-user mode.

If you don't want to use devfs, you can remove the need for it from a filesystem by copying /dev from someplace, making a bunch of /dev/ubd devices:

UML# for i in 0 1 2 3 4 5 6 7; do mknod ubd$i b 98 $[$i * 16]; done
and changing /etc/fstab and /etc/inittab to refer to the non-devfs devices.

Compiling and Installing kernel Modules

UML modules are built in the same way as the native kernel (with the exception of the 'ARCH=um' that you always need for UML):

```
host% make modules ARCH=um
```

Any modules that you want to load into this kernel need to be built in the user-mode pool. Modules from the native kernel won't work. If you notice that the modules you get are much larger than they are on the host, see the note above about the size of the final UML binary. You can install them by using ftp or something to copy them into the virtual machine and dropping them into /lib/modules/`uname -r`. You can also get the kernel build process to install them as follows:

1. with the kernel not booted, mount the root filesystem in the top level of the kernel pool:

   host% mount root_fs mnt -o loop

2. run

   host% make modules_install INSTALL_MOD_PATH=`pwd`/mnt ARCH=um

3. unmount the filesystem

   host% umount mnt

4. boot the kernel on it

If you can't mount the root filesystem on the host for some reason (like it's a COW file), then an alternate approach is to mount the UML kernel tree from the host into the UML with hostfs and run the modules_install inside UML:

1. With UML booted, mount the host kernel tree inside UML at the same location as on the host:

   UML# mount none -t hostfs *path to UML pool* -o *path to UML pool*

2. Run make modules_install:

   UML# cd *path to UML pool* ; make modules_install

The depmod at the end may complain about unresolved symbols because there is an incorrect or missing System.map installed in the UML filesystem. This appears to be harmless. insmod or modprobe should work fine at this point.

When the system is booted, you can use insmod as usual to get the modules into the kernel. A number of things have been loaded into UML as modules, especially filesystems and network protocols and filters, so most symbols which need to be exported probably already are. However, if you do find symbols that need exporting, let us know, and they'll be "taken care of".

If you try building an external module against a UML tree, you will find that it doesn't compile because of missing includes. There are less obvious problems with the CFLAGS that the module Makefile or script provides which would make it not run even if it did build. To get around this, you need to provide the same CFLAGS that the UML kernel build uses.

A reasonably slick way of getting the UML CFLAGS is

cd *uml-tree* ; make script 'SCRIPT=@echo $(CFLAGS)' ARCH=um
If the module build process has something that looks like
$(CC) $(CFLAGS) *file*
then you can define CFLAGS in a script like this
CFLAGS=`cd *uml-tree* ; make script 'SCRIPT=@echo $(CFLAGS)' ARCH=um`
and like this in a Makefile
CFLAGS=$(shell cd *uml-tree* ; make script 'SCRIPT=@echo $$(CFLAGS)' ARCH=um)

APPENDIX B: UML Networking

Setting up the network

This page describes how to set up the various transports and to provide a UML instance with network access to the host, other machines on the local net, and the rest of the net.

As of 2.4.5, UML networking has been completely redone to make it much easier to set up, fix bugs, and add new features.

There is a new helper, uml_net, which does the host setup that requires root privileges.

There are currently five transport types available for a UML virtual machine to exchange packets with other hosts:

- ethertap
- TUN/TAP
- Multicast
- a switch daemon
- slip
- slirp
- pcap

The TUN/TAP, ethertap, slip, and slirp transports allow a UML instance to exchange packets with the host. They may be directed to the host or the host may just act as a router to provide access to other physical or virtual machines.

The pcap transport is a synthetic read-only interface, using the libpcap binary to collect packets from interfaces on the host and filter them. This is useful for building preconfigured traffic monitors or sniffers.

The daemon and multicast transports provide a completely virtual network to other virtual machines. This network is completely disconnected from the physical network unless one of the virtual machines on it is acting as a gateway.

With so many host transports, which one should you use? Here's when you should use each one:

- ethertap - if you want access to the host networking and it is running 2.2
- TUN/TAP - if you want access to the host networking and it is running 2.4. Also, the TUN/TAP transport is able to use a preconfigured device, allowing it to avoid using the setuid uml_net helper, which is a security advantage.
- Multicast - if you want a purely virtual network and you don't want to set up anything but the UML

- a switch daemon - if you want a purely virtual network and you don't mind running the daemon in order to get somewhat better performance
- slip - there is no particular reason to run the slip backend unless ethertap and TUN/TAP are just not available for some reason
- slirp - if you don't have root access on the host to setup networking, or if you don't want to allocate an IP to your UML
- pcap - not much use for actual network connectivity, but great for monitoring traffic on the host

Ethertap is available on 2.4 and works fine. TUN/TAP is preferred to it because it has better performance and ethertap is officially considered obsolete in 2.4. Also, the root helper only needs to run occasionally for TUN/TAP, rather than handling every packet, as it does with ethertap. This is a slight security advantage since it provides fewer opportunities for a nasty UML user to somehow exploit the helper's root privileges.

General setup

First, you must have the virtual network enabled in your UML. If are running a prebuilt kernel from this site, everything is already enabled. If you build the kernel yourself, under the "Network device support" menu, enable "Network device support", and then the three transports.

The next step is to provide a network device to the virtual machine. This is done by describing it on the kernel command line. The general format is

eth**<n>**=**<transport>**,**<transport args>**
For example, a virtual ethernet device may be attached to a host ethertap device as follows:
eth0=ethertap,tap0,fe:fd:0:0:0:1,192.168.0.254
This sets up eth0 inside the virtual machine to attach itself to the host /dev/tap0, assigns it an ethernet address, and assigns the host tap0 interface an IP address.

Note that the IP address you assign to the host end of the tap device must be different than the IP you assign to the eth device inside UML. If you are short on IPs and don't want to comsume two per UML, then you can reuse the host's eth IP address for the host ends of the tap devices. Internally, the UMLs must still get unique IPs for their eth devices. You can also give the UMLs non-routable IPs (192.168.x.x or 10.x.x.x) and have the host masquerade them. This will let outgoing connections work, but incoming connections won't without more work, such as port forwarding from the host.

Also note that when you configure the host side of an interface, it is only acting as a gateway. It will respond to pings sent to it locally, but is not useful to do that since it's a host interface. You are not talking to the UML when you ping that interface and get a response.

You can also add devices to a UML and remove them at runtime. See the mconsole page for details.

The sections below describe this in more detail.

Once you've decided how you're going to set up the devices, you boot UML, log in, configure the UML side of the devices, and set up routes to the outside world. At that point, you will be able to talk to any other machines, physical or virtual, on the net.

If ifconfig inside UML fails and the network refuses to come up, run 'dmesg' to see what ended up in the kernel log. That will usually tell you what went wrong.

Userspace daemons

You will likely need the setuid helper, or the switch daemon, or both. They are both installed with the RPM and deb, so if you've installed either, you can skip the rest of this section.

If not, then you need to check them out of CVS , build them, and install them. The helper is uml_net, in CVS /tools/uml_net, and the daemon is uml_switch, in CVS /tools/uml_router. They are both built with a plain 'make'. Both need to be installed in a directory that's in your path - /usr/bin is recommend. On top of that, uml_net needs to be setuid root.

Specifying ethernet addresses

Below, you will see that the TUN/TAP, ethertap, and daemon interfaces allow you to specify hardware addresses for the virtual ethernet devices. This is generally not necessary. If you don't have a specific reason to do it, you probably shouldn't. If one is not specified on the command line, the driver will assign one based on the device IP address. It will provide the address fe:fd:nn:nn:nn:nn where nn.nn.nn.nn is the device IP address. This is nearly always sufficient to guarantee a unique hardware address for the device. A couple of exceptions are:

- Another set of virtual ethernet devices are on the same network and they are assigned hardware addresses using a different scheme which may conflict with the UML IP address-based scheme
- You aren't going to use the device for IP networking, so you don't assign the device an IP address

If you let the driver provide the hardware address, you should make sure that the device IP address is known before the interface is brought up. So, inside UML, this will guarantee that:
UML# ifconfig eth0 192.168.0.250 up
If you decide to assign the hardware address yourself, make sure that the first byte of the address is even. Addresses with an odd first byte are broadcast addresses, which you don't want assigned to a device.

UML interface setup
Once the network devices have been described on the command line, you should boot UML and log in.

The first thing to do is bring the interface up:

UML# ifconfig eth*n ip-address* up
You should be able to ping the host at this point.

To reach the rest of the world, you should set a default route to the host:

UML# route add default gw *host ip*
Again, with host ip of 192.168.0.4:
UML# route add default gw 192.168.0.4
This page used to recommend setting a network route to your local net. This is wrong, because it will cause UML to try to figure out hardware addresses of the local machines by arping on the interface to the host. Since that interface is basically a single strand of ethernet with two nodes on it (UML and the host) and arp requests don't cross networks, they will fail to elicit any responses. So, what you want is for UML to just blindly throw all packets at the host and let it figure out what to do with them, which is what leaving out the network route and adding the default route does.

Note: If you can't communicate with other hosts on your physical ethernet, it's probably because of a network route that's automatically set up. If you run 'route -n' and see a route that looks like this:

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
192.168.0.0	0.0.0.0	255.255.255.0	U	0	0	0	eth0

with a mask that's not 255.255.255.255, then replace it with a route to your host:
UML# route del -net 192.168.0.0 dev eth0 netmask 255.255.255.0
UML# route add -host 192.168.0.4 dev eth0
This, plus the default route to the host, will allow UML to exchange packets with any machine on your ethernet.

Multicast
The simplest way to set up a virtual network between multiple UMLs is to use the mcast transport. This was written by Harald Welte and is present in UML version 2.4.5-5um and later. Your system must have multicast enabled in the kernel and there must be a multicast-capable network device on the host. Normally, this is eth0, but if there is no ethernet card on the host, then you will likely get strange error messages when you bring the device up inside UML.

To use it, run two UMLs with

eth0=mcast
on their command lines. Log in, configure the ethernet device in each machine with different IP addresses:
UML1# ifconfig eth0 192.168.0.254
UML2# ifconfig eth0 192.168.0.253
and they should be able to talk to each other.

The full set of command line options for this transport are

eth*n*=mcast,*ethernet address,multicast address,multicast port,ttl*
Harald's original README is here and explains these in detail, as well as some other issues.

TUN/TAP with the uml_net helper

TUN/TAP is the preferred mechanism on 2.4 to exchange packets with the host. The TUN/TAP backend has been in UML since 2.4.9-3um.

The easiest way to get up and running is to let the setuid uml_net helper do the host setup for you. This involves insmod-ing the tun.o module if necessary, configuring the device, and setting up IP forwarding, routing, and proxy arp. If you are new to UML networking, do this first. If you're concerned about the security implications of the setuid helper, use it to get up and running, then read the next section to see how to have UML use a preconfigured tap device, which avoids the use of uml_net.

If you specify an IP address for the host side of the device, the uml_net helper will do all necessary setup on the host - the only requirement is that TUN/TAP be available, either built in to the host kernel or as the tun.o module. The format of the command line switch to attach a device to a TUN/TAP device is

eth**<n>**=tuntap,,,**<host IP address>**
For example, this argument will attach the UML's eth0 to the next available tap device, assign the IP address 192.168.0.254 to the host side of the tap device, and assign an ethernet address to it based on the IP address assigned to it by ifconfig inside UML.
eth0=tuntap,,,192.168.0.254

If you using the uml_net helper to set up the host side of the networking, as in this example, note that changing the UML IP address will cause uml_net to change the host routing and arping to match. This is one reason you should not be using uml_net if there is any possibility that the user inside the UML may be unfriendly. This feature is convenient, but can be used to make the UML pretend to be something like your name server or mail server, and the host will steal packets intended for those servers and forward them to the UML. See the next section for setting up networking in a secure manner.

There are a couple potential problems with running the TUN/TAP transport on a 2.4 host kernel

- TUN/TAP seems not to work on 2.4.3 and earlier. Upgrade the host kernel or use the ethertap transport.
- With an upgraded kernel, TUN/TAP may fail with
- File descriptor in bad state

  This is due to a header mismatch between the upgraded kernel and the kernel that was originally installed on the machine. The fix is to make sure that /usr/src/linux points to the headers for the running kernel.

These were pointed out by Tim Robinson in this uml-user post .

TUN/TAP with a preconfigured tap device

If you prefer not to have UML use uml_net (which is somewhat insecure), with UML 2.4.17-11, you can set up a TUN/TAP device beforehand. The setup needs to be done as root, but once that's done, there is no need for root assistance. Setting up the device is done as follows:

- Create the device with tunctl (available from the UML utilities tarball)

  host# tunctl -u *uid*

  where *uid* is the user id or username that UML will be run as. This will tell you what device was created.

- Configure the device IP (change IP addresses and device name to suit)

host# ifconfig tap0 192.168.0.254 up

- Set up routing and arping if desired - this is my recipe, there are other ways of doing the same thing

  host# bash -c 'echo 1 > /proc/sys/net/ipv4/ip_forward'

  host# route add -host 192.168.0.253 dev tap0

  host# bash -c 'echo 1 > /proc/sys/net/ipv4/conf/tap0/proxy_arp'

  host# arp -Ds 192.168.0.253 eth0 pub

  Note that this must be done every time the host boots - this configuration is not stored across host reboots. So, it's probably a good idea to stick it in an rc file. An even better idea would be a little utility which reads the information from a config file and sets up devices at boot time.

- Rather than using up two IPs and ARPing for one of them, you can also provide direct access to your LAN by the UML by using a bridge.

  host# brctl addbr br0

  host# ifconfig eth0 0.0.0.0 promisc up

  host# ifconfig tap0 0.0.0.0 promisc up

  host# ifconfig br0 192.168.0.1 netmask 255.255.255.0 up

  host# brctl stp br0 off

  host# brctl setfd br0 1

  host# brctl sethello br0 1

  host# brctl addif br0 eth0

  host# brctl addif br0 tap0

  Note that 'br0' should be setup using ifconfig with the existing IP address of eth0, as eth0 no longer has its own IP.

- Also, the /dev/net/tun device must be writable by the user running UML in order for the UML to use the device that's been configured for it. The simplest thing to do is

host# chmod 666 /dev/net/tun

Making it world-writeable looks bad, but it seems not to be exploitable as a security hole. However, it does allow anyone to create useless tap devices (useless because they can't configure them), which is a DOS attack. A somewhat more secure alternative would to be to create a group containing all the users who have preconfigured tap devices and chgrp /dev/net/tun to that group with mode 664 or 660.

- Once the device is set up, run UML with 'eth0=tuntap,*device name*' (i.e. 'eth0=tuntap,tap0') on the command line (or do it with the mconsole config command).
- Bring the eth device up in UML and you're in business.

If you don't want that tap device any more, you can make it non-persistent with
host# tunctl -d *tap device*
Finally, tunctl has a -b (for brief mode) switch which causes it to output only the name of the tap device it created. This makes it suitable for capture by a script:
host# TAP=`tunctl -u 1000 -b`

Ethertap

Ethertap is the general mechanism on 2.2 for userspace processes to exchange packets with the kernel.

To use this transport, you need to describe the virtual network device on the UML command line. The general format for this is

eth<**n**>=ethertap,<**device**>,<**ethernet address**>,<**host IP address**>
So, the previous example
eth0=ethertap,tap0,fe:fd:0:0:0:1,192.168.0.254
attaches the UML eth0 device to the host /dev/tap0, assigns it the ethernet address fe:fd:0:0:0:1, and assigns the IP address 192.168.0.254 to the host side of the tap device.

The tap device is mandatory, but the others are optional. If the ethernet address is omitted, one will be assigned to it.

The presence of the tap IP address will cause the helper to run and do whatever host setup is needed to allow the virtual machine to communicate with the outside world. If you're not sure you know what you're doing, this is the way to go.

If it is absent, then you must configure the tap device and whatever arping and routing you will need on the host. However, even in this case, the uml_net helper still needs to be in your path and it must be setuid root if you're not running UML as root. This is because the tap device doesn't support SIGIO, which UML needs in order to use something as a source of input. So, the helper is used as a convenient asynchronous

IO thread. If you're using the uml_net helper, you can ignore the following host setup - uml_net will do it for you. You just need to make sure you have ethertap available, either built in to the host kernel or available as a module.

If you want to set things up yourself, you need to make sure that the appropriate /dev entry exists. If it doesn't, become root and create it as follows:

mknod /dev/tap**<minor>** c 36 **<minor>** + 16
For example, this is how to create /dev/tap0:
mknod /dev/tap0 c 36 0 + 16
You also need to make sure that the host kernel has ethertap support. If ethertap is enabled as a module, you apparently need to insmod ethertap once for each ethertap device you want to enable. So,
host# insmod ethertap
will give you the tap0 interface. To get the tap1 interface, you need to run
host# insmod ethertap unit=1 -o ethertap1

The switch daemon
Note: This is the daemon formerly known as uml_router, but which was renamed so the network weenies of the world would stop growling at me.

The switch daemon, uml_switch, provides a mechanism for creating a totally virtual network. By default, it provides no connection to the host network (but see -tap, below).

The first thing you need to do is run the daemon. Running it with no arguments will make it listen on a default pair of unix domain sockets.

If you want it to listen on a different pair of sockets, use

-unix *control socket data socket*

If you want it to act as a hub rather than a switch, use
-hub

If you want the switch to be connected to host networking (allowing the umls to get access to the outside world through the host), use

-tap tap0

Note that the tap device must be preconfigured (see "TUN/TAP with a preconfigured tap device", above). If you're using a different tap device than tap0, specify that instead of tap0.

uml_switch can be backgrounded as follows

host% uml_switch [options] < /dev/null > /dev/null

The reason it doesn't background by default is that it listens to stdin for EOF. When it sees that, it exits.

The general format of the kernel command line switch is

eth*n*=daemon,*ethernet address,socket type,control socket,data socket*
You can leave off everything except the 'daemon'. You only need to specify the ethernet address if the one that will be assigned to it isn't acceptable for some reason. The rest of the arguments describe how to communicate with the daemon. You should only specify them if you told the daemon to use different sockets than the default. So, if you ran the daemon with no arguments, running the UML on the same machine with
eth0=daemon
will cause the eth0 driver to attach itself to the daemon correctly.

Slip

Slip is another, less general, mechanism for a process to communicate with the host networking. In contrast to the ethertap interface, which exchanges ethernet frames with the host and can be used to transport any higher-level protocol, it can only be used to transport IP.

The general format of the command line switch is

eth*n*=slip,*slip IP*
The slip IP argument is the IP address that will be assigned to the host end of the slip device. If it is specified, the helper will run and will set up the host so that the virtual machine can reach it and the rest of the network.

There are some oddities with this interface that you should be aware of. You should only specify one slip device on a given virtual machine, and its name inside UML will be 'umn', not 'eth0' or whatever you specified on the command line. These problems will be fixed at some point.

Slirp

slirp uses an external program, usually /usr/bin/slirp, to provide IP only networking connectivity through the host. This is similar to IP masquerading with a firewall, although the translation is performed in user-space, rather than by the kernel. As slirp does not set up any interfaces on the host, or changes routing, slirp does not require root access or setuid binaries on the host.

The general format of the command line switch for slirp is:

eth*n*=slirp,*ethernet address,slirp path*

The ethernet address is optional, as UML will set up the interface with an ethernet address based upon the initial IP address of the interface. The slirp path is generally /usr/bin/slirp, although it will depend on distribution.

The slirp program can have a number of options passed to the command line and we can't add them to the UML command line, as they will be parsed incorrectly. Instead, a wrapper shell script can be written or the options inserted into the ~/.slirprc file. More information on all of the slirp options can be found in its man pages.

The eth0 interface on UML should be set up with the IP 10.2.0.15, although you can use anything as long as it is not used by a network you will be connecting to. The default route on UML should be set to use 'eth0' without a gateway IP:

UML# route add default dev eth0
slirp provides a number of useful IP addresses which can be used by UML, such as 10.0.2.3 which is an alias for the DNS server specified in /etc/resolv.conf on the host or the IP given in the 'dns' option for slirp.

Even with a baudrate setting higher than 115200, the slirp connection is limited to 115200. If you need it to go faster, the slirp binary needs to be compiled with FULL_BOLT defined in config.h.

pcap

The pcap transport is attached to a UML ethernet device on the command line or with uml_mconsole with the following syntax:

eth*n*=pcap,*host interface*,*filter expression*,*option1*,*option2*
The expression and options are optional.

The interface is whatever network device on the host you want to sniff. The expression is a pcap filter expression, which is also what tcpdump uses, so if you know how to specify tcpdump filters, you will use the same expressions here. The options are up to two of 'promisc', 'nopromisc', 'optimize', 'nooptimize'. 'promisc' and 'nopromisc' control whether pcap puts the host interface into promiscuous mode. 'optimize' and 'nooptimize' control whether the pcap expression optimizer is used.

Example:

eth0=pcap,eth0,tcp
eth1=pcap,eth0,!tcp
will cause the UML eth0 to emit all tcp packets on the host eth0 and the UML eth1 to emit all non-tcp packets on the host eth0.

Setting up the host yourself
If you don't specify an address for the host side of the ethertap or slip device, UML won't do any setup on the host. So this is what is needed to get things working (the

examples use a host-side IP of 192.168.0.251 and a UML-side IP of 192.168.0.250 - adjust to suit your own network):

- The device needs to be configured with its IP address. Tap devices are also configured with an mtu of 1484. Slip devices are configured with a point-to-point address pointing at the UML ip address.

  host# ifconfig tap0 arp mtu 1484 192.168.0.251 up

  host# ifconfig sl0 192.168.0.251 pointopoint 192.168.0.250 up

- If a tap device is being set up, a route is set to the UML IP.

  UML# route add -host 192.168.0.250 gw 192.168.0.251

- To allow other hosts on your network to see the virtual machine, proxy arp is set up for it.

  host# arp -Ds 192.168.0.250 eth0 pub

- Finally, the host is set up to route packets.

  host# echo 1 > /proc/sys/net/ipv4/ip_forward

APPENDIX C: UML Utilities

Compiling and installing UML utilities

Many features of the UML kernel require a user-space helper program, so a uml_utilities package is distributed separately from the kernel patch which provides these helpers. Included within this is:

- port-helper - Used by consoles which connect to xterms or ports
- tunctl - Configuration tool to create and delete tap devices
- uml_net - Setuid binary for automatic tap device configuration
- uml_switch - User-space virtual switch required for daemon transport

The uml_utilities tree is compiled with:
host# make && make install
Note that UML kernel patches may require a specific version of the uml_utilities distribution. If you don't keep up with the mailing lists, ensure that you have the latest release of uml_utilities if you are experiencing problems with your UML kernel, particularly when dealing with consoles or command-line switches to the helper programs

APPENDIX D: Virtual Network Shell Scripts

1. Physical Linux Host Scripts

uml_start

```
#####################################################################
# Super-script used to create the overlay UML network               #
#                                                                   #
#                                                                   #
#                                    #
#####################################################################

#!/bin/bash

uml_start_usage()
{
  echo "Usage: ./uml_start [ -c | -h] [-a | -u | -v <subnet_count> <nodes_per_subnet>]
"
  echo "   -a for attacker network"
  echo "   -c for the core network"
  echo "   -u for user network"
  echo "   -v for victim network"
  echo "   -h to print this help message"
}

start_attack_network()
{
  SUBNET_IP="$1"
  SUBNET_COUNT="$2"
  NODE_COUNT="$3"

  clear
  echo 'setting up attack network'
  sleep 2
  ./subnet_builder $SUBNET_IP $SUBNET_COUNT $NODE_COUNT attack
}

start_core_network()
{
  clear
  echo 'starting netcore_builder'
  sleep 2
  ./netcore_builder
```

```
}

start_user_network()
{
  SUBNET_IP="$1"
  SUBNET_COUNT="$2"
  NODE_COUNT="$3"

  clear
  echo 'setting up user network'
  sleep 2
  ./subnet_builder $SUBNET_IP $SUBNET_COUNT $NODE_COUNT user
}

start_victim_network()
{
  SUBNET_IP="$1"
  SUBNET_COUNT="$2"
  NODE_COUNT="$3"

  clear
  echo 'setting up victim network'
  sleep 2
  ./subnet_builder $SUBNET_IP $SUBNET_COUNT $NODE_COUNT victim
}

if [ $# -eq 0 ]
then
 uml_start_usage
 return 1
fi

  SUBNET_COUNT="$2"
  NODE_COUNT="$3"
  SUBNET_IP="192.168"

# set up route for SUBNET_IP on physical network
route add -net "$SUBNET_IP.0.0/16" dev eth0 2> /dev/null

getopts a:cu:v: options

  case "$options" in

    a)  start_attack_network $SUBNET_IP $SUBNET_COUNT $NODE_COUNT ;;
    c)  start_core_network;;
    u)  start_user_network $SUBNET_IP $SUBNET_COUNT $NODE_COUNT ;;
```

```
    v)    start_victim_network $SUBNET_IP $SUBNET_COUNT $NODE_COUNT
;;
    \?)   uml_start_usage;;
  esac
```

subnet_builder

```
#!/bin/bash

if [ $# -ne 4 ]
then
        echo wrong number of arguments
        echo usage: subnet_builder subnet_IP no_of_subnets no_of_host_per_subnet
network_type
        echo example: subnet_builder 192.168.0 1 2 attack
        echo The example sets up 1 subnet per router with 2 hosts where 192.168.0 is
the subnet address
        echo 'network_type = {victim|attack|user}'
        return 1
fi


#create a router and its subnet(s)
#arguments: router_ID subnet_ip subnet_count node_count switch_port
build_subnet()
{
        if [ $# -ne 5 ]
        then
                echo wrong number of arguments
                echo usage: build_subnet router_ID subnet_IP no_of_subnets
node_count switch_port
                echo example: build_subnet 1 192.168 1 2 22000
                exit 1
        fi

ID="$1"
SUBNET_IP="$2"
SUBNET_COUNT="$3"
NODE_COUNT="$4"
PORT="$5"
ROUTER_IP="$SUBNET_IP.$ID.1"
HOST_IP=`ifconfig eth0 | grep 'inet addr' | cut -d: -f2 | cut -d' ' -f1`


i=1
```

```
j=0
k=1

#set router tap device drivers
eth_devices="eth0=tuntap,,,$ROUTER_IP"

#set up the switches
while [ $k -le $SUBNET_COUNT ]

do
 uml_switch -unix $((PORT+i)) $((PORT+1+i)) -hub < /dev/null &
 #the next line is for the router's other eth devices
 eth_devices=${eth_devices}'
eth'$((j+1))=daemon,,unix,"$((PORT+i)),$((PORT+1+i))"
 i=$((i+2))
 j=$((j+1))
 k=$((k+1))
done
echo setting up the switches ...
sleep 2 #allow things to stabilize
echo 'finished'

        # set up the router/firewall (with 2 consoles)
        linux umid="Router:$ROUTER_IP"
":$ID:$SUBNET_COUNT:$NODE_COUNT:$HOST_IP:"
ubd0=${file_systems}"/router/root_fs_cow$ID",${file_systems}'/root_fs'
${eth_devices} ssl=pty con=pty con0=xterm con1=xterm mem=64M &

        # set up the hosts
        i=1
        while [ $i -le $SUBNET_COUNT ]
        do
                j=0
                while [ $j -lt $NODE_COUNT ]
                do
                        k=$[16*(i-1)+17+j]
                        . vhost_setup "$SUBNET_IP.$ID.$k" &
                        j=$((j+1))
                done
                i=$((i+1))
        done
}

uml_home='/home/uml'
cd ${uml_home}
file_systems=${uml_home}'/filesystem'
#cleanup previous uml junks that might still be lurking around
```

```
./cleanup

SUBNET_IP="$1"
SUBNET_COUNT="$2"
NODE_COUNT="$3"
NET_TYPE="$4"

#set up the subdirectories for cow files properly
#start with the router cow file directory
        if ! test -e ${file_systems}'/router' -a -d ${file_systems}'/router'
        then
                echo "${file_systems}'/router' does not exist, creating ..."
                mkdir ${file_systems}'/router'
                chmod a+rw ${file_systems}'/router'
        fi

#check whether there is a subdirectory for all the specified sub-subnet
#if not create one
subnet_home=${file_systems}'/subnet'
i=1
while [ $i -le $SUBNET_COUNT ]
do
        if ! test -e ${subnet_home}${i} -a -d ${subnet_home}${i}
        then
                echo "${subnet_home}${i} does not exist, creating ..."
                mkdir ${subnet_home}${i}
                chmod a+rw ${subnet_home}${i}
        fi
        i=$((i+1))
done

# enable ip forwarding proxy arping etc on the physical host
sysctl -w net.ipv4.ip_forward="1"
sysctl -w net.ipv4.conf.all.proxy_arp="1"
route add -net 192.168.0.0/16 dev eth0

ID=0
if [ $NET_TYPE = victim ]
then

        ID=1
        PORT=22000
        build_subnet $ID $SUBNET_IP $SUBNET_COUNT $NODE_COUNT
$PORT

elif [ $NET_TYPE = user ]
then
```

```
        ID=9
        PORT=22000
        build_subnet $ID $SUBNET_IP $SUBNET_COUNT $NODE_COUNT
$PORT

        ID=10
        PORT=23000
        build_subnet $ID $SUBNET_IP $SUBNET_COUNT $NODE_COUNT
$PORT

elif [ $NET_TYPE = attack ]
then

        ID=11
        PORT=22000
        build_subnet $ID $SUBNET_IP $SUBNET_COUNT $NODE_COUNT
$PORT

        ID=12
        PORT=23000
        build_subnet $ID $SUBNET_IP $SUBNET_COUNT $NODE_COUNT
$PORT

fi
```

Vhost_setup

```
#!/bin/bash
# set up the virtual uml host

if [ $# != 1 ]
then
        echo
        echo wrong number of arguments
        echo usage: vhost_setup ip_address
        echo example: vhost_setup 192.168.0.33
        echo
        return
fi
#extract the ip
full_ip="$1"
subnet_ip=`echo "$1" | cut -d'.' -f1-3`
ip_add=`echo "$1" | cut -d'.' -f4`
HOST_IP=`ifconfig eth0 | grep 'inet addr' | cut -d: -f2 | cut -d' ' -f1`
ROUTER_ID=`echo "$1" | cut -d'.' -f3`
```

```
P=2
case "$ROUTER_ID" in
        1|2|9|11)       P=2;;
        10|12)          P=3;;
esac

case "$ip_add" in

        1[7-9]|2[0-9]|30)
        subsubnet='subnet1';switch="2"$P"001,2"$P"002";;
        3[3-9]|4[0-6])
        subsubnet='subnet2';switch="2"$P"003,2"$P"004";;
        49|5[0-9]|6[0-2])
        subsubnet='subnet3';switch="2"$P"005,2"$P"006";;
        6[5-9]|7[0-8])
        subsubnet='subnet4';switch="2"$P"007,2"$P"008";;
        8[1-9]|9[1-4])
        subsubnet='subnet5';switch="2"$P"009,2"$P"010";;
        9[7-9]|10[0-9]|110)
        subsubnet='subnet6';switch="2"$P"011,2"$P"012";;
        11[3-9]|12[0-6])
        subsubnet='subnet7';switch="2"$P"013,2"$P"014";;
        129|13[0-9]|14[1-2])
        subsubnet='subnet8';switch="2"$P"015,2"$P"016";;
        14[5-9]|15[0-8])
        subsubnet='subnet9';switch="2"$P"017,2"$P"018";;
        16[1-9]|17[0-4])
        subsubnet='subnet10';switch="2"$P"019,2"$P"020";;
        17[7-9]|18[0-9]|190)
        subsubnet='subnet11';switch="2"$P"021,2"$P"022";;
        19[3-9]|20[0-6])
        subsubnet='subnet12';switch="2"$P"023,2"$P"024";;
        209|21[0-9]|22[0-2])
        subsubnet='subnet13';switch="2"$P"025,2"$P"026";;
        22[5-9]|23[0-8])
        subsubnet='subnet14';switch="2"$P"027,2"$P"028";; #note tap and router are
on this subnet

        *)      echo
                echo 'the ip address you entered is not in the valid range'
                echo 'usage: vhost_setup ip_address'
                echo 'example: vhost_setup 192.168.0.33 '
                return
                ;;
esac
```

```
#start the virtual umlinux hosts
root_file_location='/home/uml/filesystem/root_fs'
cow_file_location='/home/uml/filesystem/'${subsubnet}'/cow_fs_for_ip_'${full_ip}
linux umid="VHost:${subnet_ip}.${ip_add}" "::::$HOST_IP:"
ubd0=${cow_file_location},${root_file_location} eth0=daemon,,unix,${switch}
ssl=pty con=pty con0=xterm mem=64M &
```

Netcore-builder

```
#!/bin/bash

#####################################################################
# builds the core of the network with 6 routers                     #
# each router has 3 interfaces one connected to                     #
# a tunnel and the others a (UML) switch                            #
#                                                                   #
#####################################################################

Netcore_Builder_Usage() {

   echo "Usage: netcore_builder"
}

clear


SUBNET_IP="192.168"
#get ip of localhost
UML_HOME='/home/uml'
FILE_SYSTEMS=$UML_HOME'/filesystem'
cd $UML_HOME


./cleanup

if ! test -e ${FILE_SYSTEMS}'/router' -a -d ${FILE_SYSTEMS}'/router'
        then
                echo "${FILE_SYSTEMS}'/router' does not exist, creating ..."
                mkdir ${FILE_SYSTEMS}'/router'
                chmod a+rw ${FILE_SYSTEMS}'/router'
fi

# switches to connect routers
uml_switch -unix 23001 23002 < /dev/null &
uml_switch -unix 23003 23004 < /dev/null &
```

```
for i in 3 4 5 6 7 8
#for i in 6 7 8
do
  case $i in
    3|4|5) SWITCH="daemon,,unix,23001,23002";;
    6|7|8) SWITCH="daemon,,unix,23003,23004";;
  esac

  case $i in
    3|6) SWITCH="$SWITCH eth2=$SWITCH";;
  esac

        ID="$i"
        TAP_IP="$SUBNET_IP.$ID.1"

        ETH_DEVICES="eth0=tuntap,,,$TAP_IP eth1=$SWITCH"

  #start up core router
  linux umid="CRouter:$TAP_IP" ":$ID:"
ubd0="$FILE_SYSTEMS/router/root_fs_cowCRouter$i,$FILE_SYSTEMS/root_fs"
$ETH_DEVICES ssl=pty con=pty con0=xterm mem=64M &

done
```

cleanup

```
#!/bin/bash
#utility to clean up all the "mess" associated with uml

#sweep off all umlinux instances
echo 'removing umlinux instances'
kill -9 `ps aux | grep linux | cut -c10-14` 2>/dev/null
rm -rf /tmp/uml/*

echo 'removing the switches'
#remove the uml switches
kill -9 `ps aux | grep uml_switch | cut -c10-14` 2>/dev/null
rm -rf /home/uml/22*
rm -rf /home/uml/23* #and the associated files

#and the tap devices if they exist
echo 'bringing down the tap devices'
i=0
while [ $i -lt 20 ]
do
```

```
 ifconfig eth0:$i down 2>/dev/null
 tunctl -d tap$i >/dev/null 2>/dev/null
 ip t d tunl$i 2>/dev/null
 i=$((i+1))
done

echo 'finished'
```

UML host scripts

umlboot

```
#! /bin/sh

# /etc/init.d/umlboot: initial startup configuration for uml
#################################################################
#                                                               #
# This is the start-up script used inside UML to                #
# automatically configure uml host during boot up.              #
#                                                               #
# INSTALL:                                                      #
# A symlink to this file should be put under /etc/init.d        #
# with permission 0755.                                         #
# All scripts are kept in /etc/scripts                          #
# Also, setup symlinks under /etc/rcN.d (run:)                  #
#   update-rc.d -f umlboot start 5 2 3 4 5 .                    #
#                                                               #
#################################################################


HOSTNAME=`cut -d: -f1 /proc/cmdline | cut -d"=" -f2`
UML_IP=`cut -d: -f2 /proc/cmdline`
HOST_IP=`cut -d: -f6 /proc/cmdline`

# Configurable options


case "$1" in
      start)
                  #set up hostname and domain name and command prompt
                  echo "setting hostname to $HOSTNAME at domain
cs.ucl.ac.uk"
                  /sbin/sysctl -w kernel.hostname=$HOSTNAME
                  /sbin/sysctl -w kernel.domainname='cs.ucl.ac.uk'

                  if [ -e /etc/profile_backup ]
                  then
                     cp /etc/profile_backup /etc/profile

                  else
            cp  /etc/profile /etc/profile_backup
                  fi
                  echo "export PS1=$HOSTNAME" >> /etc/profile
                  echo "export HOST_IP=$HOST_IP UML_IP=$UML_IP
HOSTNAME=$HOSTNAME" >> /etc/profile
```

```
                ;;


      stop)
                ;;
      reload|force-reload)
                ;;
      restart)
                ;;

      *)
                echo "Usage: /etc/init.d/umlboot
{start|stop|reload|force-reload|restart}"
                exit 1

esac

exit 0
```

umlnet

```
#!/bin/bash

# /etc/init.d/umlnet: network configuration for uml
#################################################################
#                                                               #
# This is the start-up script used inside UML to                #
# automatically configure networking during boot up.            #
#                                                               #
# INSTALL:                                                      #
# A symlink to this file should be put under /etc/init.d        #
# with permission 0755.
      #
# All scripts are kept in /etc/scripts                          #
# Also, setup symlinks under /etc/rcN.d (run:)                  #
#   update-rc.d -f umlnet start 40 2 3 4 5 .                          #
#                                                               #
#################################################################

#UML_CONFIG_STR
UML_VHOST_STR="VHost"
UML_ROUTER_STR="Router"
UML_CROUTER_STR="CRouter"



# find out the type of this instance
get_host_type() {
   TYPE=`cut -d: -f1 /proc/cmdline | cut -d"=" -f2`
   echo $TYPE
}


config_host() {
    println "UML detected as $UML_VHOST_STR"
```

```
       /etc/scripts/vhost.conf
}

config_router() {
            println "UML detected as $UML_ROUTER_STR"
            /etc/scripts/router.conf
}

config_core()
{
    println "UML detected as Core Router"
    /etc/scripts/c_router.conf
}
println() {
    echo "umlnet: $*"
}

start_uml_net_config() {

    # extract type from umid
    uml_type=`get_host_type`

   case "$uml_type" in
            $UML_ROUTER_STR)
                config_router
                ;;
            $UML_VHOST_STR)
                config_host
                ;;
            $UML_CROUTER_STR)
                   config_core
                   ;;
      *)
          echo "unknown host type $uml_type"
          exit 1
          ;;
    esac
}

case "$1" in
    start)
      start_uml_net_config
        ;;
    *)
      ;;
esac
```

Create_tunnel

```
#!/bin/bash

# /etc/scripts/create_tunnel : tunnel creation utility
#######################################
#                                     #
# Sample:                             #
```

```
#                                    #
# TUNNEL_NAME="tunl1"                #
# REMOTE_GW= "192.168.1.2"           #
# REMOTE_TUN_IP= "192.168.2.2"       #
# LOCAL_TUN_IP="192.168.1.3"         #
#                                    #
######################################


TUNNEL_NAME=$1
REMOTE_GW=$2
REMOTE_TUN_IP=$3
LOCAL_TUN_IP=$4
PORT=$5

openvpn --remote $REMOTE_GW --dev $TUNNEL_NAME --ifconfig
$LOCAL_TUN_IP $REMOTE_TUN_IP --port $PORT --shaper 60000 &
```

vhost.conf

```
#!/bin/bash
#Virtual host configuration file

#extract ip address
ip_add=`cut -d: -f2 /proc/cmdline` #of this host
subnet_ip=`echo $ip_add | cut -d'.' -f1-3`
last_octet=`echo $ip_add | cut -d'.' -f4`
broadcast_ip=$(((last_octet & 240)|15)) # netmask 240
gateway_ip=$((broadcast_ip-1))

#configure eth0
ifconfig eth0 ${ip_add} netmask 255.255.255.240 broadcast
${subnet_ip}.${broadcast_ip} up

#create route to the router
route add default gw ${subnet_ip}.${gateway_ip}

#networking complete
```

router.conf

```
#!/bin/bash
# /etc/scripts/router.conf: router configuration utility

WD=`pwd`
cd /etc/scripts

#extract IP addresses
ip_add=`cut -d: -f2 /proc/cmdline` #of the router
ID=`cut -d: -f3 /proc/cmdline` #Router ID
```

```
subnet_count=`cut -d: -f4 /proc/cmdline`
node_count=`cut -d: -f5 /proc/cmdline`
subnet_address=`echo $ip_add | cut -d'.' -f1-3`
last_octet=`echo $ip_add | cut -d'.' -f4`
broadcast_ip=$(((last_octet & 240)|15)) # netmask 240
SUB_16ADD=`echo $ip_add | cut -d'.' -f1-2`
IP_ADD="$SUB_16ADD.$ID.2"

#bring up the eth0 driver
ifconfig eth0 $IP_ADD up

i=1
j=1
if [ $ID -eq 1 ]
then
      #j=2
      #ifconfig eth1 "$SUB_16ADD.2.2" up
      ./create_tunnel tunl1 $SUB_16ADD.3.2 $SUB_16ADD.100.3
$SUB_16ADD.100.1 5000

      sleep 5
      for IND in 3 4 5 9 10
      do
          route add -net "$SUB_16ADD.$IND.0/24" dev tunl1
      done
      ./create_tunnel tunl2 $SUB_16ADD.6.2 $SUB_16ADD.100.6
$SUB_16ADD.100.2 5100
      sleep 5
      for IND in 6 7 8 11 12
      do
          route add -net "$SUB_16ADD.$IND.0/24" dev tunl2
      done

elif [ $ID -eq 9 ]
then
      ./create_tunnel tunl1 $SUB_16ADD.4.2 $SUB_16ADD.100.4
$SUB_16ADD.100.9 5000
      sleep 5
      for IND in 1 2 3 4 5 10
      do
          route add -net "$SUB_16ADD.$IND.0/24" dev tunl1
      done
elif [ $ID -eq 10 ]
then
      ./create_tunnel tunl1 $SUB_16ADD.5.2 $SUB_16ADD.100.5
$SUB_16ADD.100.10 5000
      sleep 5
      for IND in 1 2 3 4 5 9
      do
          route add -net "$SUB_16ADD.$IND.0/24" dev tunl1
      done
elif [ $ID -eq 11 ]
then
      ./create_tunnel tunl1 $SUB_16ADD.7.2 $SUB_16ADD.100.7
$SUB_16ADD.100.11 5000
      sleep 5
      for IND in 1 2 6 7 8 12
      do
```

```
          route add -net "$SUB_16ADD.$IND.0/24" dev tunl1
      done
elif [ $ID -eq 12 ]
then
      ./create_tunnel tunl1 $SUB_16ADD.8.2 $SUB_16ADD.100.8
$SUB_16ADD.100.12 5000
      sleep 5
      for IND in 1 2 6 7 8 11
      do
          route add -net "$SUB_16ADD.$IND.0/24" dev tunl1
      done
fi

while [ $i -le $subnet_count ]
do
      router_ip=$(( 16*(i-1) + 30 )) #addresses we reserved for the
router, just below broadcast address
      ifconfig eth${j} ${subnet_address}.${router_ip} netmask
255.255.255.240 broadcast ${subnet_address}.$((router_ip + 1)) up
      i=$((i+1))
      j=$((j+1))
done

#add the route to neighbouring router(s) through eth0
case $ID in
    1)
    route add -host  "$SUB_16ADD.3.2" dev eth0
    route add -host  "$SUB_16ADD.6.2" dev eth0
    ;;
    9)
    route add -host  "$SUB_16ADD.4.2" dev eth0
    ;;
    10)
    route add -host  "$SUB_16ADD.5.2" dev eth0
    ;;
    11)
    route add -host  "$SUB_16ADD.7.2" dev eth0
    ;;
    12)
    route add -host  "$SUB_16ADD.8.2" dev eth0
    ;;*)echo 'invalid ID'

esac

# enable ip forwarding
sysctl -w net.ipv4.ip_forward="1"
# set up proxy arp for the subnets
sysctl -w net.ipv4.conf.all.proxy_arp="1"

# create a route to the physical host
HOST_IP=`cut -d: -f6 /proc/cmdline`
HOST_NET=`echo $HOST_IP | cut -d'.' -f1-3`
route add -net $HOST_NET.0/24 dev eth0
cd "$WD"
```

c_router.conf

```
#!/bin/bash
# /etc/scripts/c_router.conf: core router configuration utility

ID=`cut -d: -f3 /proc/cmdline`
WD=`pwd`

cd /etc/scripts

case $ID in

   3) ifconfig eth0 192.168.3.2 up
      ifconfig eth1 192.168.34.1 up
      ifconfig eth2 192.168.35.1 up
      ./create_tunnel tunl1 192.168.1.2 192.168.100.1 192.168.100.3
5000
      sleep 5
      route add -net 192.168.1.0/24 dev tunl1
      route add -net 192.168.2.0/24 dev tunl1
      route add -net 192.168.4.0/24 gw 192.168.34.1
      route add -net 192.168.5.0/24 gw 192.168.35.1
      route add -net 192.168.9.0/24 gw 192.168.34.1
      route add -net 192.168.10.0/24 gw 192.168.35.1
      route add -host 192.168.1.2 dev eth0;;

   4) ifconfig eth0 192.168.4.2 up
      ifconfig eth1 192.168.34.2 up
      ./create_tunnel tunl1 192.168.9.2 192.168.100.9 192.168.100.4
5000
      sleep 5
      route add -net 192.168.9.0/24 dev tunl1
      route add -net 192.168.1.0/24 gw 192.168.34.2
      route add -net 192.168.3.0/24 gw 192.168.34.2
      route add -net 192.168.5.0/24 gw 192.168.34.2
      route add -net 192.168.2.0/24 gw 192.168.34.2
      route add -host 192.168.9.2 dev eth0;;

   5) ifconfig eth0 192.168.5.2 up
      ifconfig eth1 192.168.35.2 up
      ./create_tunnel tunl1 192.168.10.2 192.168.100.10 192.168.100.5
5000
      sleep 5
      route add -net 192.168.10.0/24 dev tunl1
      route add -net 192.168.1.0/24 gw 192.168.35.2
      route add -net 192.168.3.0/24 gw 192.168.35.2
      route add -net 192.168.4.0/24 gw 192.168.35.2
      route add -net 192.168.2.0/24 gw 192.168.35.2
      route add -host 192.168.10.2 dev eth0;;

   6) ifconfig eth0 192.168.6.2 up
      ifconfig eth1 192.168.67.1 up
      ifconfig eth2 192.168.68.1 up
      ./create_tunnel tunl1 192.168.1.2 192.168.100.2 192.168.100.6
5100
```

```
      sleep 5
      route add -net 192.168.1.0/24 dev tunl1
      route add -net 192.168.2.0/24 dev tunl1
      route add -net 192.168.7.0/24 gw 192.168.67.1
      route add -net 192.168.8.0/24 gw 192.168.68.1
      route add -net 192.168.11.0/24 gw 192.168.67.1
      route add -net 192.168.12.0/24 gw 192.168.68.1
      route add -host 192.168.1.2 dev eth0;;

   7) ifconfig eth0 192.168.7.2 up
      ifconfig eth1 192.168.67.2 up
      ./create_tunnel tunl1 192.168.11.2 192.168.100.11 192.168.100.7
5000
      sleep 5
      route add -net 192.168.11.0/24 dev tunl1
      route add -net 192.168.1.0/24 gw 192.168.67.2
      route add -net 192.168.2.0/24 gw 192.168.67.2
      route add -net 192.168.6.0/24 gw 192.168.67.2
      route add -net 192.168.8.0/24 gw 192.168.67.2
      route add -host 192.168.11.2 dev eth0;;

   8) ifconfig eth0 192.168.8.2 up
      ifconfig eth1 192.168.68.2 up
      ./create_tunnel tunl1 192.168.12.2 192.168.100.12 192.168.100.8
5000
      sleep 5
      route add -net 192.168.12.0/24 dev tunl1
      route add -net 192.168.1.0/24 gw 192.168.68.2
      route add -net 192.168.2.0/24 gw 192.168.68.2
      route add -net 192.168.6.0/24 gw 192.168.68.2
      route add -net 192.168.7.0/24 gw 192.168.68.2
      route add -host 192.168.12.2 dev eth0;;

   *) echo 'error in network configuration';exit 1;;
esac

#enable ip forwarding
sysctl -w net.ipv4.ip_forward="1"

#enable proxy arping
sysctl -w net.ipv4.conf.all.proxy_arp="1"
cd "$WD"
```